공학석사 학위논문

애플리케이션 인식 기반 동적 메모리 요청 조절을 통한 메모리 간섭 지연시간 개선

Application-Aware Dynamic Memory Request Throttling to Reduce Memory Interference Latency

2018년 8월

서울대학교 융합과학기술대학원

융합과학부 지능형융합시스템 전공

함 대 식

초 록

현대의 멀티 코어 프로세서는 최신 아키텍처를 통해 제한된 전력 예산 내에서 고성능을 달성할 수 있기 때문에 다양한 미션 크리티컬 임베디드 시스템에 빠르게 채택되고 있다. 이러한 미션 크리티컬 임베디드 시스템에서는 특정 작업을 전용으로 수행하는 많은 저 사양 마이크로 컨트롤러가 소수의 고성능 멀티 코어 프로세서로 통합되고 있다. 이와 같이 통합된 미션 크리티컬 임베디드 시스템은 다양한 컴퓨팅 자원을 공유하는 응용들의 자원 경합에 대한 관리가 필요하다. 자원 경합에 대한 관리가 효과적으로 되지 않으면 미션 크리티컬한 응용이 다른 응용들 과의 간섭으로 지연되어 큰 사고로 이어질 수도 있다.

중요도가 다른 애플리케이션을 보유한 임베디드 시스템의 대표적인 사례 중 하나는 InfoADAS이다. ADAS는 첨단 운전자 보조 시스템으로 중요도가 높은 응용이 주로 수행되고, 인포테인먼트는 단지 승객에게 정보와 편의를 제공하는 일반 응용이다. ADAS에는 카메라(Camera), 레이더(Radar), 라이다(LiDAR)로부터 수집된 다량의 데이터를 처리해서 정보를 얻는 데이터 집약적인 응용이 많다. 이런 응용을 수행하는 시스템에서는 메모리 경합을 효과적을 차단하는 것이 필수적이다.

본 논문에서는 메모리 경합에 의한 중요 응용의 end-to-end latency 지연 문제를 해결하는 애플리케이션 인식 기반 동적 메모리 요청 비율 throttling 기법을 제안한다. 본 기법은 시스템내 수행 중인 응용을 중요 응용과 일반 응용으로 분리한 후 cgroup으로 관리한다. 주기적으로

메모리 경합이 발생했는지 예측하고 경합이 발생했을 경우 CPUFreq 거버너의 CPU frequency 조절을 통해 일반 응용의 메모리 요청을 제한한다. 기법의 효용성을 검증하기 위해 CPU-GPU 멀티코어 아키텍처를 갖고 있는 NVIDIA Jetson TX2 보드에서 Linux kernel 4.4.38을 탑재하고 실험을 진행하였다. 실험 결과 기법 적용 전 대비 10.6%의 성능 개선 효과를 확인하였고, 그에 따른 런 타임 오버헤드는 미비하였다.

주요어 : 메모리 경합, 메모리 간섭, CPU-GPU 이기종 멀티코어, CPU throttling, 중요도 혼합 응용 시스템,

학 번 : 2016-26589

목 차

그림 목차

표 목차

제 1 장 서 론

최근 많은 임베디드 시스템에서 딥 뉴럴 네트워크를 동작시키는 AI응용이 수행된다. 그 중 대표적인 응용이 자율 주행차에서의 ADAS응용이다. ADAS 첨단 운전자 보조 시스템으로 적응형 크루즈 컨트롤, 사각 지대 모니터링, 차선 이탈 경고, 나이트 비전, 차선 유지 보조 및 충돌 경고 시스템과 자동조향 및 브레이크 조작 등이 포함된다. ADAS가 제공하는 기능을 수행하기 위해서는 많은 데이터를 이용해 많은 연산을 제한된 시간내에 처리해야 한다. 카메라나 라이다, 레이다를 통해 사람, 사물 등의 오브젝트(object)를 디텍션(detection)하고 그 결과를 바탕으로 제동 및 조향을 한다. 정확한 인식을 위해서 카메라, 각종 센서를 통해 많은 데이터를 수집하고, 수집한 데이터를 주로 딥 뉴럴 네트워크를 통해 처리한다. 이렇게 많은 데이터를 주어진 시간내에 처리하기 위해서는 큰 컴퓨팅 용량이 필요하다.

ADAS와 같은 응용을 다루는 미션 크리티컬(mission-critical) 임베디드 시스템에서는 낮은 컴퓨팅 능력을 가지면서 특정 응용만 전용으로 처리하는 마이크로 컨트롤러를 다수 사용하기 보다는, 컴퓨팅 능력이 충분한 멀티코어 플랫폼을 적용하는 사례가 늘고 있다[3]. 이런 흐름은 미션 크리티컬한 임베디드 시스템에서 요구하는 고사양의 컴퓨팅 용량과 제한된 예산의 전력을 만족시키는 SoC가 개발되었기에 가능해졌다[1]. 이런 SoC에서는 고성능 저전력의 요구사항을 만족시키기 위해 주로 GPU를 탑재해 사용한다. GPU는 CPU대비 저전력으로 동일한 병렬 연산을 빠르게 처리할 수 있기 때문이다. 멀티코어

플랫폼으로 인해 응용들의 성능은 향상되었지만 서로 공존함으로써 여러가지 문제가 발생할 수 있다.

자율주행 자동차 시스템에서는 ADAS를 위한 미션 크리티컬한 응용 외에 중요도가 덜한 인포테인먼트(infortainment) 응용도 수행된다. 인포테인먼트 응용은 네이게이션, 동영상 스트리밍 서비스 등과 같이 사용자에게 편의 정보와 오락을 제공한다. ADAS응용과 인포테인먼트 응용이 공존하는 시스템을 InfoADAS시스템이라고 하고, 두 응용을 한 번에 InfoADAS응용이라 지칭한다. 두 종류의 응용은 서로 중요도가 다르다. ADAS 응용은 미션 실패(failure)가 발생되면 사고로 이어질 수 있기 때문에 더 중요한 응용으로 분류된다. 보통 ADAS 같이 안전에 필수적인 기능을 수행하는 응용은 타이밍 요구 사항을 갖고 있고, 이런 응용은 대부분의 경우 실시간 시스템에서 동작한다[3]. 반면, 인포테인먼트 응용은 실패해도 인명 피해를 주는 사고로 이어지지 않기 때문에 중요하지 않는 응용으로 분류된다. 이렇게 중요도가 다른 두 응용이 한 시스템내에서 수행되면 서로 공유하는 자원의 경합이 발생하게 된다. 미션 크리티컬한 응용이 공유 자원을 사용하려 할 때 비중요 응용의 선점으로 인해 자원 사용을 하지 못하면 지연이 발생하게 된다. 이로 인해 중요 응용이 만족해야 하는 요구사항을 만족시키지 못할 수 있고 이는 큰 문제로 이어질 수 있다.

딥러닝 응용과 같이 데이터를 많이 사용하는 응용을 동작시키는 시스템에서 자원 경합 문제는 특히 메모리에서 많이 발생된다. 메모리 경합 문제는 메모리에 요청을 보내는 요청자들이 하나의 메모리를 공유함으로써 발생된다. 요청자들은 주로 CPU 코어, GPU, 이외의 DMA 장치들이 있다. 싱글 코어 시스템에서도 CPU와 DMA(Direct Memory Access) 컨트롤러 사이에서 발생되지만 멀티코어 시스템에서 코어

간에도 발생되고, 컴퓨팅 성능을 증가시키기 위해 등장한 CPU-GPU가 포함된 이기종의 멀티코어 시스템에서는 더욱 두드러진다[4]. 이렇게 메모리 경합으로 중요 응용의 메모리 요청 처리 시간이 지연되면 중요 응용의 양 종단간 수행시간(end-to-end latency)을 지연시키게 된다.

메모리 경합문제를 해결하기 위해 기존 연구자들은 메모리를 공간적으로 분리시키거나 시간적으로 분리시키려 하였다. 공간적으로 분리시키는 것은 메모리 컨트롤러와 메모리 사이를 연결하는 여러 개 채널을 응용별로 할당하거나, 하나의 채널내의 메모리를 구성하는 뱅크들을 응용 별로 할당하는 것이다[7][8][9]. 이렇게 전용 공간을 응용 별로 할당하면 응용의 메모리 요청양에 따라 많이 사용되지 않는 공간이 생기게 되고 이는 메모리 사용률 저하를 초래한다. 또한, 전용으로 나눌 수 있는 공간대비 응용의 수가 많으면 전용 공간을 할당하는 것이 불가능해 여전히 경합이 발생한다.

시간적으로 분리하려는 연구는 주로 메모리 컨트롤러에서 메모리 요청의 처리 순서를 응용의 특징에 따라 정하는 연구들이다[10][11][12][14][15]. 중요 응용과 비 중요 응용으로 나눠 우선순위 스케줄링(scheduling)을 하도록 메모리 컨트롤러를 수정하는 기법과 응용 별 공정하게 메모리 접근 시간을 갖도록 공정 분배 스케줄링을 하도록 메모리 컨트롤러를 수정하는 기법 등이 있다. 하지만 이는 메모리 컨트롤러에 복잡한 로직 변경을 해야 하고 운영체제에서 전달하는 정보를 저장하는 공간을 필요로 하므로 COTS(Commercial Off-The-Shelf) 시스템에서는 적합하지 않다.

본 논문에서는 메모리 경합에 의한 중요 응용의 end-to-end latency 지연 문제를 해결하기 위해 비 중요 응용이 보내는 메모리

요청을 throttling하는 기법을 제안하고자 한다. 그리고 제안하는 기법의 효과를 실험적 검증을 통해 증명한다. 본 장의 1절에서는 이 문제를 풀고자 했던 지난 연구들을 살펴봄으로써 우리 연구를 진행하게 된 동기에 대해 설명하고, 2절에서는 본 논문의 내용에 대해 간략히 설명한다. 3절에서는 본 논문의 구성에 대해 설명한다.

제 1 절 연구 동기

싱글 코어에서 성능향상을 위해 멀티코어가 도입되었다. 하지만 코어간 메모리를 공유하는 아키텍처를 갖는 시스템의 경우 메모리 경합에 의한 태스크 수행시간 지연 문제가 발생되었다. 이를 해결하기 위해 많은 연구들이 진행되었다. 진행된 연구들은 크게 두분류로 나뉜다.

첫째는 여러 개의 채널 혹은 뱅크를 가지고 있는 메모리를 물리적 공간으로 나눠 각 태스크별 혹은 코어별로 전용 공간을 할당하는 것이다[7][8][9]. 메모리는 하나 혹은 여러 개 채널로 구성되고, 한 개의 채널은 하나 혹은 여러 개의 뱅크로 구성된다. 한 개의 채널에 연결되어 있는 뱅크의 집합을 랭크라고 한다. 경합을 피하기 위해 채널 파티셔닝(partitioning)을 하면 전용 채널에 속한 메모리 공간에는 해당 응용의 데이터만 저장되고 해당 응용의 메모리 요청만 전달된다. 뱅크를 파티셔닝하는 경우도 동일한 방식이다. 이 기법은 두가지 단점이 있다. 첫째, 메모리 사용률의 문제이다. 공간을 분리해 각 태스크를 공간별로 맵핑 했을 경우 그 공간을 충분히 사용하는 태스크가 있고, 공간이 부족한 태스크가 있고, 공간을 충분히 사용하지 않는 태스크가 있을 것이다. 공간이 부족한 태스크가 있음에도 불구하고 공간이 남는

태스크로 인해 메모리 사용률이 떨어지는 경우가 발생한다. 이때 공간이 부족한 태스크는 메모리에 저장하지 못한 데이터를 디스크로 보내는 스왑(swap)이 필요하다. 디스크로 보내어진 데이터에 접근할 때 메모리 페이지 폴트(page fault)가 발생되어 성능이 떨어진다. 둘째, 파티셔닝 할 수 있는 공간 수 대비 동시에 수행중인 태스크의 수가 많으면 하나의 공간에 여러 개의 태스크가 할당될 수밖에 없어 여전히 메모리 경합이 발생한다.

메모리 경합을 해결하기 위한 연구 방향 두번째는 공간적이 아닌 시간적으로 분리해서 메모리를 사용하는 것이다. 이 방식을 사용하면 위에서 언급한 두가지의 단점은 없어진다. 시간적으로 분리하는 방법은 크게 메모리 요청을 스케줄링 하는 방법과 메모리 요청을 throttling하는 방법으로 나뉜다. 메모리 요청 스케줄링은 메모리 컨트롤러에 전달된 메모리 요청 처리 순서를 정하는 것이다[10][11][12][14][15]. 관련 연구들은 주로 응용의 중요도나 특징에 따라 메모리 요청순서를 정하는 방식이다. 메모리 요청 스케줄링 정책은 메모리 컨트롤러 내에서 구현된다. 메모리 요청 throttling은 메모리 요청을 보내는 주체인 CPU 혹은 GPU등의 메모리 요청자가 요청하는 메모리 요청 빈도를 조절하는 방법이다[16][17][20].

메모리 요청 throttling기법은 운영체제에서 구현 가능하다. 반면 메모리 요청 스케줄링 정책을 변경하기 위해선 메모리 컨트롤러를 수정해야 한다. 이는 일반적으로 복잡한 하드웨어 로직을 도입해야 하고, 메모리 요청의 우선순위와 같이 스케줄링을 위해 운영체제에서 전달받은 정보를 메모리 컨트롤러 내에 저장해야 하므로 추가적인 저장 공간을 요구한다. 그래서 COTS(Commercial Off-The-Shelf) 시스템에서는 사용할 수 없다. 때문에 우리는 운영체제에서 간편하게 구현 가능한

메모리 요청 throttling 방법을 이용해 메모리 경합에 의한 지연 시간을 감소시키는 연구를 진행하였다.

제 2 절 연구 내용

본 연구의 목적은 메모리 경합에 의해 중요 응용의 end-to-end latency를 감소시키는 것이다. 본 논문에서는 이 목표를 달성하기 위해 애플리케이션 인식 기반 동적 메모리 요청 throttling 기법을 제안한다. 본 기법은 시스템내에 여러 응용이 동시에 수행되어 메모리 경합이 발생할 때 중요 응용과 일반 응용을 인식하고 일반 응용이 메모리 컨트롤러에 보내는 메모리 요청의 비율을 조정함으로써 메모리 경합 시 중요 응용이 겪을 실행 시간 지연을 감소시키는 기법이다. 이 기법을 실현시키기 위해서는 다음 세가지 동작이 필요하다.

- 중요 응용과 일반 응용을 인식
- 메모리 경합이 발생할 것인지 예측
- 메모리 요청자의 메모리 요청 비율을 throttling

첫 번째, 중요 응용과 일반 응용을 인식하기 위해 본 논문에서는 critical task chain이라는 개념을 도입한다[21]. Critical task chain내에 있는 태스크를 critical task group, 그렇지 않은 태스크를 normal task group으로 분류하는 방법을 제공한다.

두 번째, 본 논문은 중요 응용의 메모리 경합에 의한 지연시간을 줄이기 위한 연구이다. 때문에 본 논문에서의 메모리 경합은 중요 응용이 수행되고 있을 때에 발생한 경합만으로 제한한다. 경합이 발생할

것이지 예측하기 위해선 현재 시스템 내에 수행중인 태스크가 critical task group에 속했는지 normal task group에 속했는지를 판별해야 한다. 그리고 메모리 컨트롤러의 메모리 요청 버퍼 내에 있는 아직 처리되지 않은 메모리 요청 수를 바탕으로 앞으로 메모리 경합의 정도를 예측하는 방법을 제안한다.

세 번째 메모리 요청자의 메모리 요청 비율을 throttling 하기 위해 본 논문에서는 일반 응용을 수행하는 메모리 요청자를 판별해 메모리 요청 비율을 컨트롤 하는 방법을 제안하다.

본 연구의 실험에서는 실제 자율주행차에 사용되고 있는 NVIDIA Driver PX2에 탑재된 것과 동일한 SoC가 탑재되어 있는 NVIDIA Jetson TX2 보드를 사용하였다. 미션 크리티컬 임베디드 시스템의 대표적인 예인 자율주행 시스템에서 본 기법의 효용성을 증명하기 위해서이다. Jetson TX2는 ARM CPU와 NVIDIA GPU가 탑재된 이종의 멀티코어 시스템이고 공유 메모리를 사용하고 있다. 소프트웨어는 CUDA 8.0 버전의 GPGPU 라이브러리와 리눅스 4.4.38 운영체제를 탑재했다. 널리 사용되는 오브젝트 디텍션 응용인 YOLO를 가지고 실험한 결과 제안한 기법을 적용한 경우 그렇지 않은 경우에 비해서 10.6%의 end-to-end latency 개선효과를 확인하였다. 제한된 기법의 오버헤드는 1.76% 정도로 크지 않음을 확인하였다. 이는 본 기법이 중요 응용의 메모리 경합에 의한 지연 시간을 줄이는데 효과적인 기법임을 보여준다.

제 3 절 논문 구성

본 학위 논문의 나머지 부분은 다음과 같이 구성된다. 2장에서는 배경 지식과 기존 연구의 동향을 살펴본다. 3장에서는 본 연구의 대상 시스템 모델을 설명하고, 풀고자 하는 문제를 설명한다. 4장에서는 문제의 해결방안인 동적 메모리 요청 throttling 기법에 대해 설명한다. 5장에서는 다양한 실험을 통해 제안된 기법의 효용성을 확인한다. 마지막으로 6장에서는 본 연구의 결론을 맺는다.

제 2 장 배경 지식과 관련 연구

본 장에서는 본 연구를 이해하기 위해 필요한 배경 지식과 본 연구를 하게 된 동기를 마련해준 관련 연구에 대해 설명한다. 우선 본 논문의 실험에 사용되는 GPU를 탑재한 NVIDIA Jetson TX2를 통해 GPU에 대한 소개와 CPU-GPU간 공유 메모리 구조 및 사용 가능한 메모리 동작 옵션에 대해 설명한다. 그리고 본 논문에서 제안하는 기법의 구현에 필요한 도구이자 Linux에서 제공하는 자원관리 도구인 cgroup에 대해 설명한다. 이어서 메모리 경합을 해결하기 위한 기존 연구들에 대해 소개한다.

제 1 절 CPU-GPU 이기종 멀티코어 SoC

최근 딥러닝 응용이 사용되는 경우가 많아 제한된 전력 한계 내에서 더 많은 연산 능력이 요구되었고 그에 부합하는 GPU를 탑재하는 시스템이 증가하고 있다[1]. 그 중 NVIDIA Driver PX2는 자율주행 시스템을 위한 대표적인 CPU-GPU 이기종 아기텍쳐 시스템이다. Driver PX2의 Tegra SoC의 구성은 크게 CPU와 GPU 그리고 공유메모리로 되어있다. 본 논문의 대상 하드웨어인 NVIDIA Jetson TX2도 동일한 SoC를 사용하는 보드이다. 본 절에서는 먼저 CPU-GPU 이기종 멀티코어 SoC인 Jstson TX2의 구성요소 중 가장 주목받는 GPU에 대한 소개를 먼저 하고, GPU의 프로그래밍 언어인 CUDA와 CPU-GPU간 메모리 동작 옵션 그리고 Jetson TX2의 하드웨어 구조에 대해 설명한다.

NVIDIA GPU는 게이밍과 서버 컴퓨팅 시장에 사용되는 GPU의 대표 주자이자, 최근에는 자율주행 AI 시장을 위한 GPU의 선두 주자이다. 그럼에도 불구하고 GPU의 내부 동작에 대해서는 공식적으로 알려진 바가 거의 없다. 일부 문서를 통해 프로그램 코딩을 할 수 있도록 도와주고, 몇 가지 가용 가능한 옵션을 설명해주는 정도이다[1]. 그래서 GPU에서 현재 동작 중인 태스크가 어떤 것인지 정확히 확인이 불가능하다. 4장에서 본 연구의 기법 중 현재 수행중인 태스크를 식별하는 방법 설명 시 GPU에서 동작 중인 태스크를 보수적으로 확인하는 방법이 소개될 것이다. NVIDIA GPU는 또한 여러 개의 태스크가 GPU 사용 요청 시 그 요청들을 스케줄링 하는 방법을 정확히 알 수 없다. 정책 공개가 안되어 있고, 새로운 정책을 적용 불가 하므로 실시간 시스템에서 GPU를 사용하는데 제약이 있다.

NVIDIA GPU는 CUDA라는 GPGPU 프로그래밍 언어를 통해 GPU를 동작 시킬 수 있다[18]. 이는 NVIDIA가 자체 개발했다. CUDA 프로그램의 전형적인 구조는 다음과 같다.

① GPU 메모리 할당
② CPU 메모리에서 GPU 메모리로 입력 데이터 복사
③ 커널(kernel)이라고 불리는 GPU에서 병렬로 수행될 함수 런치(launch)
④ 수행 결과를 CPU 메모리에서 GPU 메모리로 복사
⑤ 할당했던 GPU 메모리 해제

CPU와 GPU 간 데이터 복사 시 사용가능한 메모리 복사 동작은 세가지이다. 첫째 traditional memory는 CPU 메모리에서 GPU 메모리로 복사하고 캐쉬 메모리를 사용해 데이터에 접근할 수 있다.

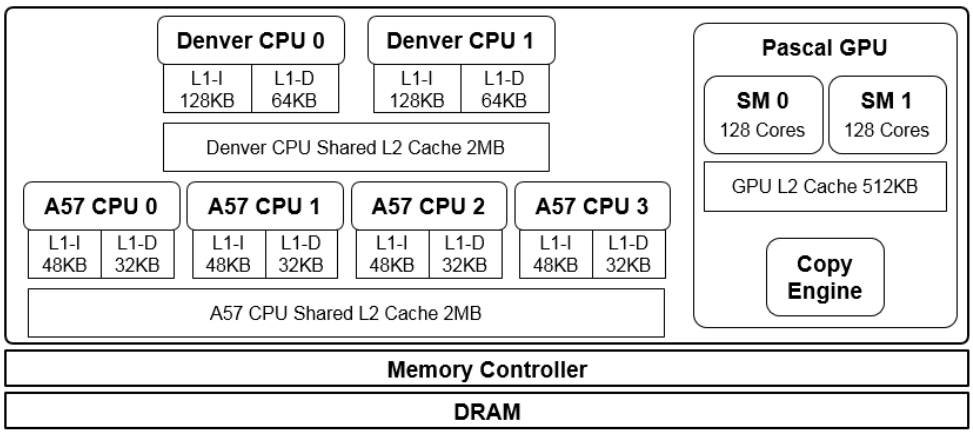

그림 1. NVIDIA Jetson TX2의 하드웨어 구성

둘째 zero-copy memory는 CPU와 GPU 사이의 복사없이 CPU 메모리 영역을 하나의 포인터를 사용해 GPU에서도 접근 가능하다. 하지만 일관성을 위해 캐쉬 메모리를 사용할 수 없다. 셋째 unified memory는 메모리 복사를 디바이스 드라이버가 자동으로 해준다. Unified memory를 사용할 경우 프로그래머 관점에서는 CPU와 GPU가 하나의 메모리를 공유하는 것처럼 보여 져서 프로그램 상에서 복사하는 동작이 필요 없다. 프로그램 코딩 시 복사 동작 없이 하나의 포인터를 이용할 수 있기 때문에 코딩이 쉬어 진다.

그림 1처럼 NVIDIA Jetson TX2는 크게 CPU 클러스터 2개와 GPU 1개 그리고 하나의 공유 메모리로 구성된다[19]. CPU 클러스터는 각각 ARMv8 2GHz Denver 코어 2개와 ARMv8 2GHz A57 코어 4개로 구성되어 있다. 두 아키텍처가 같은 주파수이지만 Denver 아키텍처가 파이프라이닝(pipelining)등의 최적화로 속도가 더 빠르다. GPU는 Pascal 아키텍처로 2개의 Streaming Multiprocessor로 되어 있고 하나에 128개의 코어로 구성되어 있다. GPU는 메모리를 CPU와 공유하는지 여부에 따라 discrtet GPU와 intergrated GPU로 나뉠 수 있다. Jetson TX2는 CPU와 메모리를 공유하는 integreted GPU이다.

메모리는 8GB용량의 1.866GHz DRAM이다. Jetson TX2는 NVIDIA Driver PX2(Autocruise)와 동일하게 5~15W정도의 power를 소모한다. 이는 1~2kW정도의 power를 생산하는 보통의 자동차 시스템에 사용되기 적당한 용량이다.

제 2 절 cgroup 소개

본 절에서는 cgroup의 역할 소개와 간단한 예를 통해 본 연구의 해결책에서 사용된 cgroup에 대한 이해를 돕는다. cgroup은 프로세스들의 자원의 사용을 제한하고 격리시키는 Linux 커널 기능이다. 이를 통해 개발자는 CPU 코어, 메모리, 네트워크 대역폭과 같은 자원을 실행중인 개발자 정의 그룹 간에 할당할 수 있다. 여기서 그룹을 생성하고 관리하는 역할을 cgroup 코어가 담당하고 자원을 할당해 주는 역할을 cgroup 컨트롤러가 담당한다. cgroup은 트리 구조를 형성한다. 시스템의 모든 프로세스는 하나의 단일 cgroup에 속한다. 프로세스의 모든 스레드는 동일한 cgroup에 속한다. 프로세스 추가 및 생성 시 생성된 자식 프로세스는 부모 프로세스의 cgroup에 속한다. 프로세스를 다른 cgroup으로 이주시킬수 있다. 프로세스의 이주는 이미 존재하는 자식 프로세스에 영향을 미치지는 않는다[5].

실제 예를 통해 cgroup이 어떻게 사용되는지 살펴보자. 메모리 대역폭을 cgroup별로 할당하고 싶다면 우선 커널 설정 시 원하는 서브시스템을 활성화해야 한다. 여기서 서브시스템은 해당 cgroup 컨트롤러의 동작을 위해 필요한 시스템을 말한다. 활성화를 하고

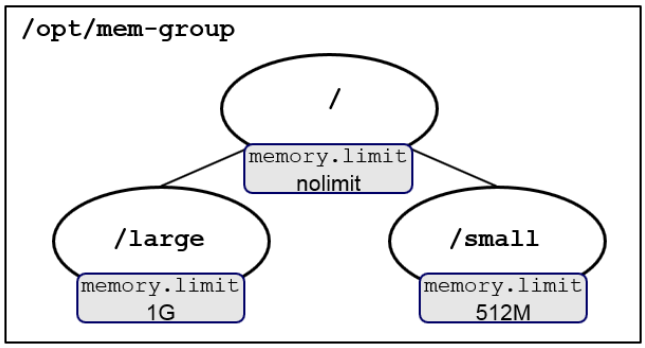

그림 2. cgroup 계층 구조

마운트를 해야 cgroup을 생성할 수 있다. 사용가능한 서브시스템은 /proc/cgroups에서 확인할 수 있다. 마운트 시 사용하고자 하는 서브시스템과 그룹명을 명시한다. cgroup은 리눅스 파일시스템으로 관리된다. mem-group라는 그룹명으로 마운트 하면 /opt/mem-group 디렉토리에 해당 그룹이 생성된다. 이 디렉터리에 해당 group의 메모리를 할당량을 지정할 수 있는 memory.limit 이라는 파일이 자동으로 생성된다. 이 파일은 그룹을 추가로 생성하기 위해 디렉터리를 만들면 그 안에 자동으로 생성된다. 그룹별로 메모리 사용량을 관리하기 위해 /mem-group 디렉터리에 /large와 /small 디렉터리를 생성하고 각각 1GB와 512MB의 메모리만 사용할 수 있게 설정하면 해당 디렉터리에 고유 식별자가 있는 프로세스들은 설정된 대역폭만큼만 메모리를 사용할 수 있게 된다. 이와 같은 구조를 그림2에 표현하였다.

제 3 절 관련 연구

본 절에서는 메모리 경합문제를 해결하기 위해서 기존의 연구들이 어떤 방법을 사용했는지 살펴본다. 멀티코어 시스템에서 코어 간에 서로 공유하는 메모리의 경합문제를 해결하는 방법은 크게 두가지로 나눌 수

있다. 하나의 메모리 자원을 공간적으로 분리하는 것이고 나머지 하나는 시간적으로 분리하는 것이다.

3.1 메모리 자원을 공간적으로 분리하는 방법

공간적 분리는 쓰레드 혹은 코어에 전용 메모리 공간을 물리적으로 할당하여 경합을 막는 방법이다. 이렇게 하면 동시에 여러 쓰레드가 메모리에 접근할 수 있는 이점이 있다. 전용 공간을 할당하는 방법에는 전용 채널을 할당하는 방법과 전용 뱅크를 할당하는 방법이 있다.

[7]에서는 응용별로 전용 채널을 할당하는 방법을 제안했다. 런타임에 응용들의 메모리 접근 강도를 조사하여 강도가 높은 응용과 강도가 낮은 응용으로 분리한다. 채널 또한 두 그룹으로 나누고 채널을 할당 받길 원하는 응용이 속한 그룹에 할당된 채널들 중 선호되는 채널에 각 응용을 할당하는 방식이다. 선호되는 채널은 로드가 적은 채널이다. 메모리 접근 강도가 높은 응용은 한 번의 분류 작업을 더 거친다. 로우 버퍼 지역성이 높은 응용과 그렇지 않은 응용으로 분류되어 서로 다른 채널을 할당 받는다. [7]은 메모리 요청 스케줄링도 고려하였다. 메모리 접근 강도가 아주 적고 짧은 메모리 접근 시간을 요구하는 응용의 경우 전용 채널을 할당하지 않고 가장 먼저 수행될 수 있도록 스케줄링 한다. 하지만 이 방법은 이미 모든 채널이 다른 응용에 의해 점유되고 있을 때는 의미가 없다.

[8][9]에서는 전용 뱅크를 할당하는 방법을 제안했다. [8]에서는 동적으로 코어마다 특정 뱅크를 할당하는 기법을 제안했다. 둘 이상의 코어 간에 하나의 뱅크를 공유함으로써 발생하는 간섭을 문제로 보고 코어 간의 뱅크 공유를 제거하는 동적인 기법을 제안했다. [9]에서는

쓰레드들의 뱅크 경합에 의한 성능지연 문제에 초점을 맞추고 있다. 한 쓰레드당 8~16개 뱅크에 해당하는 크기의 메모리를 할당해주면 쓰레드의 성능에 영향을 주지 않는다는 관찰을 바탕으로 쓰레드에 뱅크를 할당해준다. 뱅크를 여러 개 그룹으로 나누고 각 쓰레드를 특정 그룹에 할당하는 방식이다. 그리고 로우 버퍼(row buffer) 힛-미스(hit-miss) 비율로 제안하는 기법의 효용성을 평가한다.

이렇게 공간적으로 메모리를 분리해 쓰레드 혹은 코어 별 전용 메모리 공간을 사용하는 방식은 두가지 단점이 있다. 첫째는 메모리 사용율의 문제이다. 공간을 분리해 각 쓰레드를 공간별로 맵핑 했을 경우 그 공간을 충분히 사용하는 쓰레드가 있고, 공간이 부족한 쓰레드가 있고, 공간을 충분히 사용하지 않는 쓰레드가 있을 것이다. 공간을 충분히 사용하지 않는 쓰레드로 인해 메모리 사용률이 저하된다. 또한 공간이 부족한 쓰레드는 디스크로의 스왑 데이터 양이 많아져 메모리 접근 성능이 저하되는 문제가 있다. 둘째는 여전히 존재하는 메모리 경합의 발생 가능성이다. 시스템상의 모든 응용 또는 쓰레드 수가 전용 채널 혹은 뱅크 수보다 많은 경우에는, 완전히 독립적으로 할당하는 것이 불가능하다. 이런 경우에는 하나의 채널 혹은 뱅크를 두고 여러 응용 혹은 쓰레드가 경합을 할 수밖에 없다.

3.2 메모리 자원을 시간적으로 분리하는 방법

시간적 분리는 한 번에 여러 쓰레드가 메모리에 접근 시 한 번에 하나의 쓰레드만 메모리에 접근할 수 있게 하는 방법이다. 시간적 분리를 하면 하나의 쓰레드가 메모리 전체를 사용할 수 있는 이점이 있다. 시간적으로 분리하는 방법에는 메모리 요청을 스케줄링 하는 방법과 throttling하는 방법으로 나뉜다.

■ 메모리 요청 스케줄링

메모리 요청 스케줄링은 주로 메모리 컨트롤러에서 수행된다. 주로 사용되는 메모리 요청 스케줄링 방식은 FIFO, 라운드 로빈, FR-FCFS이다. FR-FCRS는 뱅크의 로우 버퍼 힛트 율(hit ratio)을 향상시키는 방향으로 메모리 요청을 스케줄링 한다[10]. 로우 버퍼에 로드 되어 있는 데이터에 접근하는 메모리 요청을 우선적으로 처리한다. 만약 로우 버퍼에 없는 데이터를 접근할 경우 해당 데이터가 위치한 메모리 영역에 접근해 로우 버퍼에 로드 하는 과정이 추가되어 메모리 접근 속도가 느려 진다.

[11]에서는 중요 응용과 일반 응용이 혼합된 실시간 시스템에서 중요 응용이 보낸 메모리 요청이 허용되는 최대 지연 시간(WCET)을 만족시키는 범위에서 일반 응용의 메모리 접근 성능을 최대한 보장하는 것을 목표로 하는 새로운 메모리 컨트롤러의 스케줄링 기법을 제안하였다. 이를 위해서 메모리 요청 간의 우선순위를 메모리 컨트롤러에서 인지하고 우선순위 스케줄링과 선제적(preemptive) 스케줄링이 가능하도록 했다.

[12]에서는 시스템 전체의 처리율 향상을 목적으로 하는 메모리 요청 스케줄링 기법을 제안했다. 응용이 하는 일을 CPU 연산 부분과 메모리 요청 부분으로 나눌 때 CPU 연산이 많은 응용의 메모리 요청을 먼저 처리시키는 것이 핵심 아이디어이다. [11]과 동일하게 우선순위 스케줄링이 가능하도록 했다.

[14]에서는 메모리 컨트롤러가 메모리 사용 시간을 쓰레드 간에 공정하게 배분하기 위한 기법을 제안한다. 이를 위해 저자는 메모리

요청 대기로 인해 발생하는 메모리 스톨(stall) 시간을 같게 만들도록 스케줄링 해야 한다. [15]에서는 3D 애니메이션의 재생 속도를 보장하기 위해 CPU-GPU 이기종의 시스템에서 GPU 접근의 중요도를 기준으로 메모리 요청을 스케줄링 하는 기법을 제안했다.

메모리 컨트롤러에 의한 메모리 스케줄링 기법을 적용하기 위해서는 메모리 컨트롤러를 수정해야한다. 이런 방식은 일반적으로 복잡한 하드웨어 로직을 도입해야 하고, 메모리 요청의 우선순위와 같이 스케줄링을 위해 운영체제에서 전달받은 정보를 메모리 컨트롤러 내에 저장해야 하므로 추가적인 저장 공간을 요구한다. 그러므로 COTS(commercial off-the-shelf) 시스템에서는 적용하기 힘들다는 단점이 있다.

■ **메모리 요청 throttling**

메모리 요청 throttling은 메모리 요청이 메모리 컨트롤러로 보내어지기 전에 메모리 요청자의 메모리 요청 빈도를 조절하여 경합을 줄이는 방법이다. 메모리 요청자는 CPU, GPU, 기타 DMA 장치가 있다. [16]의 저자는 메모리 대역폭을 태스크 또는 CPU 코어 별로 보장해주는 방식을 제안했다. 태스크 또는 코어 별로 할당된 메모리 대역폭을 보장해주기 위해 할당된 대역폭에 따라 현재 수행 중인 코어의 수행을 계속하거나 중지시킨다. 할당된 대역폭을 만족한 태스크가 수행중인 코어는 그렇지 못한 태스크의 할당된 대역폭을 보장하기 위해 수행을 중단하는 방식이다.

[17]은 실시간 시스템에서 비 실시간 응용과의 메모리 경합에 의해 실시간 응용의 수행 시간이 지연되는 것을 막고자 한다. 개발자가 성능

요구사항 만족을 시켜야 하는 응용의 코드에 집중적으로 메모리 요청을 보내는 부분을 명시해주면 이 코드를 수행 중일 때 다른 태스크를 수행 중인 CPU 수행을 중지시키는 기법을 제안했다.

[20]에서는 CPU-GPU 이기종의 멀티코어 시스템에서 GPU에서 발생되는 메모리 요청의 처리 시간 지연을 막기 위해 CPU의 수행을 조절한다. CPU 코어에 주기적으로 주어진 메모리 대역폭 할당량을 다 소비하면 CPU 코어의 수행을 중지시켜서 GPU에서 발생되는 메모리의 간섭을 최소화하는 것을 목적으로 한다.

위 연구들은 대부분 코어 별로 메모리 대역폭을 측정해서 주어진 대역폭을 다 사용하면 해당 코어를 throttling 하는 방식이다. 대부분의 시스템에서 LLC를 공유하기 때문에 코어별로 정확한 메모리 대역폭 사용량을 구하는 것이 어렵다. 또한 경합이 발생된 시점에서 동적으로 throttling을 하는 것이 아니라 주어진 예산을 다 사용 시 정적으로 throttling을 수행하므로 과도한 throttling으로 인해 시스템 사용률이 저하될 수 있다. [17]과 같이 응용 별 메모리 요청 집중영역을 인식해 메모리 경합을 피하도록 스케줄링 하는 방식은 유용하나, 메모리 요청 집중 영역을 확인하는 것이 사실상 어렵다. [17]은 코드 상에 직접 명시해서 메모리 요청 집중 영역을 확인하는 방식을 사용했다. 하지만 이 방식은 런타임에 캐쉬 효과로 인해 정확한 경합 시점을 알려주지 못한다.

[22]에서는 동적으로 메모리 요청을 throttling하는 헤라클레스라는 해법을 제안했다. 헤라클레스는 이전연구들과 달리 메모리 경합만이 아니라 CPU, 캐쉬, 네트워크, power 등의 경합을 종합적으로 관리한다. 헤라클레스의 목적은 응용들을 LC(Latency Critical)응용 그룹과

BE(Best Effrot)응용 그룹으로 분류해, LC응용의 주어진 성능 요구사항을 만족시키면서 최대한 BE응용을 수행시켜 클라우드 서버에서 시스템 전체 사용률을 높이는 것이다. 이를 위해 메모리를 공간적으로 분리하는 방법과 시간적으로 분리하는 방법을 모두 사용하였다. 캐쉬 경합 해소를 위해서는 캐쉬를 두 공간으로 나누어 응용그룹별로 전용 공간을 할당해 주는 기법을 사용하였다. 메모리 경합 해소를 위해서는 사용중인 메모리 대역폭을 측정하고 그룹 별 사용량을 예측해서 그 양에 따라 BE응용의 CPU 코어 사용 수를 throttling하는 기법을 적용하였다. 이때 CPU 코어 사용 수를 throttling하기 위해 cgroup의 cpuset 서브시스템을 이용하였다. 헤라클레스는 응용 그룹별로 사용하는 메모리 대역폭을 알기 위해 오프라인 예측 모델을 사용하였다. 하지만 이것으로는 여러 변수가 있는 런타임에서 정확한 대역폭을 알 수 없다.

헤라클레스는 메모리 경합을 동적으로 확인해서 cgroup을 통해 throttling을 한다는 컨셉이 우리 연구와 동일하나, 세가지 부분에서 차이가 있다. 첫째, 헤라클레스는 메모리 경합을 메모리 대역폭의 포화 여부로 판단한다. 하지만 우리 기법은 경합을 비교적 정확하게 예측하는 척도로 outstanding memory request를 사용한다. 이 척도의 적합성은 4장에서 설명한다. 둘째, 헤라클레스는 메모리 요청을 throttling하기 위해 코어의 수를 조절한다. 이 방법은 태스크 이주 오버헤드가 크고, 코어수가 적은 임베디드 시스템에서는 적합하지 않을 수 있다. 또한 이기종의 멀티코어 시스템에서 사용할 경우 코어 별 성능이 다르기 때문에 코어 수를 제어하는 방식으로는 원하는 양만큼의 throttling 못할 수 있다. 그래서 우리는 코어의 frequency를 조절해서 메모리 요청을 throttling한다. 이를 위해 기존의 cgroup을 확장하여 CPU의 frequency를 조절할 수 있도록 cgroup 컨트롤러를 추가하고,

CPUFreq 거버너를 수정하였다. 셋째, 헤라클레스는 런타임에 LC응용 그룹과 BE응용 그룹을 분류하는 방법은 제시하고 있지 않다. 우리는 유저 라이브러리 함수와 수정된 커널 함수 통해 그룹별로 분류하고 관리하는 방법을 제공한다. 또한 우리는 CPU-GPU 이기종의 멀티코어 시스템에서 본 기법을 사용할 수 있도록 GPU에서 수행중인 응용을 확인하는 방법을 추가하였다.

제 3 장 문제 설명과 해결 방안 개관

본 장은 풀고자 하는 문제를 명확하게 설명하고 그 문제의 해결 방안을 간략히 설명한다. 메모리 접근 경합 문제는 시스템 내의 여러 개 코어가 하나의 메모리를 공유함으로써 발생된다. 때문에 공유 메모리를 갖는 시스템의 정확한 모델링이 필요한다. 1절에서는 대상 시스템 모델인 공유 메모리 시스템에서 메모리 요청이 어떻게 처리되는지 설명한다. 2절에서는 본 연구에서 풀고자 하는 문제를 설명한다. 3절에서는 제안하는 기법에 대해 간략히 설명한다.

제 1 절 대상 시스템 모델

본 절에서는 메모리 공유하는 시스템을 모델링한다. 본 논문의 대상이 되는 시스템은 그림 3과 같이 여러 CPU, GPU, 기타 DMA 장치들이 하나의 메모리 컨트롤러를 공유하는 이기종의 멀티 코어 시스템이다. 이 시스템은 크게 네 가지 컴포넌트로 구성되어 있다.

- 메모리 요청자
- 메모리 컨트롤러
- 외부 메모리 컨트롤러
- 메모리

메모리 요청자는 메모리 작업을 직접 메모리 컨트롤러에 전달하는 주체로서 CPU 또는 GPU, DMA 장치들이 될 수 있다. 본 연구에서는

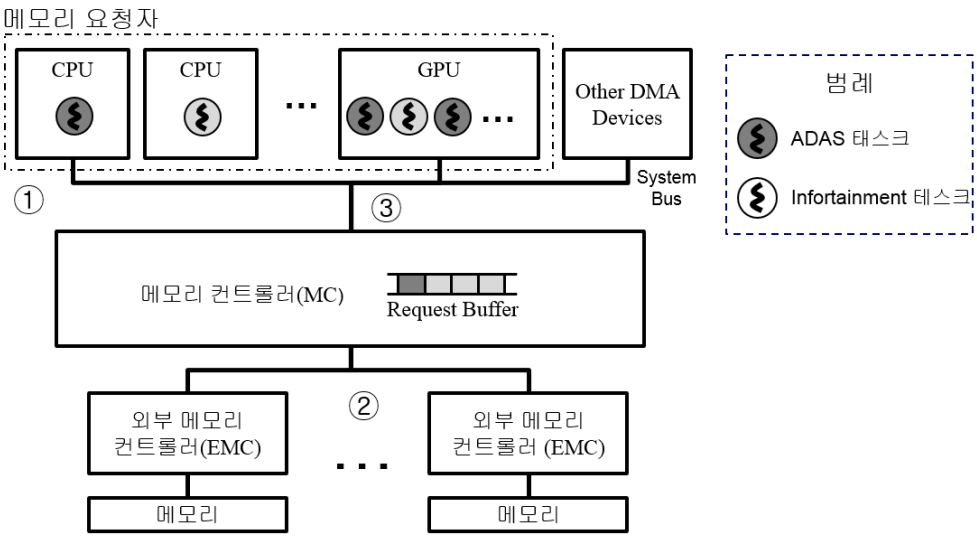

그림 3. 대상 시스템 모델

메모리 요청자를 CPU와 GPU만으로 한정한다. 메모리 컨트롤러는 메모리 요청자들로부터 전달받은 메모리 작업을 외부 메모리 컨트롤러에 전달한다. 메모리 컨트롤러는 메모리 요청 전달 순서를 정하는 스케줄링을 담당하고 아직 처리되지 않은 메모리 요청이 대기할 수 있는 메모리 요청 버퍼를 가지고 있다. 외부 메모리 컨트롤러는 실제 물리 메모리에서 데이터를 읽어오거나 쓰는 작업을 메모리에 지시하고 결과를 메모리 컨트롤러에 리턴(return)하는 역할을 한다. 메모리는 실제 메모리 동작의 주체이고, 로우 버퍼(row buffer)를 갖고 있다. 로우 버퍼는 캐쉬 메모리와 같은 역할을 한다. 로우 버퍼에는 이전 작업 데이터가 저장되어 있는데 만약 다음 작업에서 동일 로우(row) 주소로 접근 시 빠르게 데이터에 접근할 수 있게 해준다.

메모리 요청을 처리하는 순서는 다음과 같다.

① 메모리 요청자가 메모리 요청을 메모리 컨트롤러에게 전달한다.

② 메모리 컨트롤러가 메모리 요청을 외부 메모리 컨트롤러로 전달해서 메모리의 데이터를 읽거나 쓴다.

③ 결과를 메모리 컨트롤러로부터 리턴 받는다.

이때 중요 응용이 요청한 메모리 작업이 메모리 컨트롤러에서 바로 외부 메모리 컨트롤러로 전달되지 않으면, 중요 응용의 수행시간이 지연될 수 있다. 메모리 요청이 메모리 컨트롤러에서 외부 메모리 컨트롤러로 바로 전달되지 않는 원인은 일반 응용의 요청에 의해 먼저 도착한 메모리 요청들이 처리되는 시간 때문이다.

제 2 절 문제 설명

본 절에서는 제안하는 동적 메모리 요청 throttling 기법이 해결하려는 문제를 명확히 설명한다. 먼저 시스템 내에 있는 태스크들을 critical 태스크와 normal 태스크로 나누기 위해 필요한 개념인 critical task chain에 대해 설명한 후 메모리 간섭 지연(MIL, Memory Interference Latency)에 대해 정의하고 이를 바탕으로 해결하고자 하는 문제를 설명한다.

정의 1. (CRITICAL TASK CHAIN) critical task chain은 `start-of-critical-execution` 함수 호출로 불리는 태스크로부터 시작하여 `end-of-critical-execution` 함수 호출로 불리는 태스크로 끝나는 태스크 수행의 시퀀스이다.

여기서 start-of-critical-execution 함수와 end-of-critical-execution 함수는 본 기법이 제공하는 유저 라이브러리 함수이다. 그림 4와 같이 사용자가 중요하다고 여기는 코드 섹션의 시작과 끝에 명시만 하면 커널이 이를 인식해서 두 함수 사이에 생성되는 태스크 혹은 해당 태스크 수행에 연관이 있는 태스크들을 하나의 그룹으로 묶어서 관리할 수 있도록 해준다. 자세한 사항은 4장 동적 메모리 요청 throttling 기법 설명 시 다루어 진다.

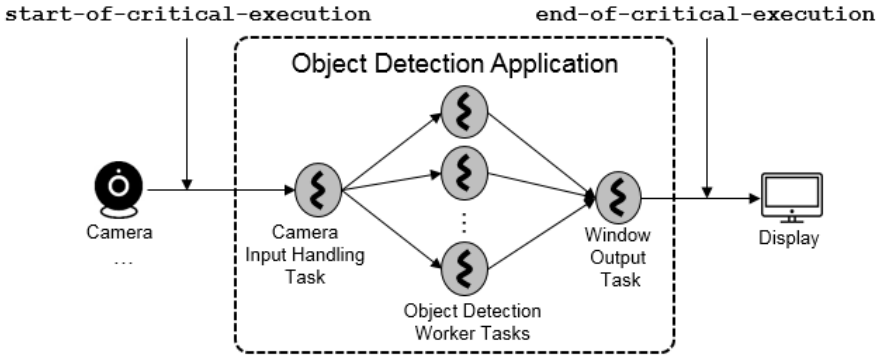

그림 4. Critical Task Chain 생성 과정

정의 2. (CRITICAL TASK GROUP) critical task group C는 critical task chain에 속해 있는 태스크들의 집합이다.

정의 3. (NOMAL TASK GROUP) normal task group N은 시스템 내에 존재하는 태스크들 중 critical task chain에 속해 있지 않은 태스크들의 집합이다.

정의 4. (MEMORY INTERFERENCE LATENCY) 태스크 $\tau \in C$ 에 의한 메모리 요청 r 이 주어졌을 때 r 의 memory interference latency(MIL)는 $C \cup N$ 에 속한 태스크들과 τ 가 함께 수행될 때의 메모리 접근 시간과 오직 C에 속한 태스크들과 τ가 함께 수행될 때의 메모리 접근 시간의 차이이다. 이때 r의 MIL을 i_r로 부른다.

$$i_r = (memory\ access\ time_r\ with\ C \cup N) - (memory\ access\ time_r\ with\ C)$$

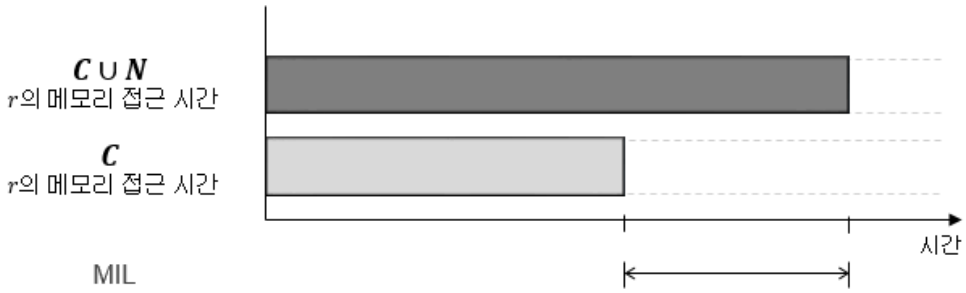

그림 5. MIL의 도식화

앞선 정의들을 바탕으로 본 학위 논문에서 다루고자 하는 문제는 다음과 같다.

문제 설명. i_r을 줄여 critical task chain의 end-to-end latency를 줄인다.

제 3 절 해결 방안 개관

Critical task chain의 메모리 경합에 의한 end-to-end latency를 줄이기 위해서는 i_r 을 줄여야 한다. 본 연구진이 제안하는 해결책의 핵심 아이디어는 i_r 의 예측량에 따라 동적으로 태스크 $\tau \in N$ 가 발생시키는 메모리 요청 비율을 throttling하는 것이다. 이를 위해서는 다음 네 가지 작업이 필요하다.

- 태스크 그룹화
- MIL 예측
- 메모리 요청 비율 감소
- 메모리 요청 비율 원복

태스크 그룹화는 critical task chain을 생성해 critical task group과 normal task group으로 나누는 작업이다. Critical task chain은 사용자가 코드 중 중요한 섹션을 새로 디자인한 라이브러리 함수를 이용해 명시해 줌으로써 생성된다. 또한 수행 중에 해당 섹션 수행에 관여하는 태스크들을 자동으로 critical task chain넣고 고유의 식별자를 커널이 관리하는 자료 구조에 저장한다. 식별자를 바탕으로 cgroup을 사용해 critical cgroup과 normal cgroup으로 나눠 관리한다.

MIL 예측은 주기적으로 현재 수행중인 태스크의 criticality와 메모리 컨트롤러의 request buffer에 아직 처리되지 않은 채 남아 있는 outstanding memory request수 측정을 통해 수행된다. 예측된 결과를 바탕으로 MIL이 발생하였으면 메모리 요청 비율을 감소시키고 발생하지 않았으면 감소시킨 메모리 요청 비율을 원복 시킨다.

이런 일련의 작업들은 그림 6과 같이 Linux 커널에서 구현된 다음 세가지 핵심 컴포넌트들에 의해 수행된다.

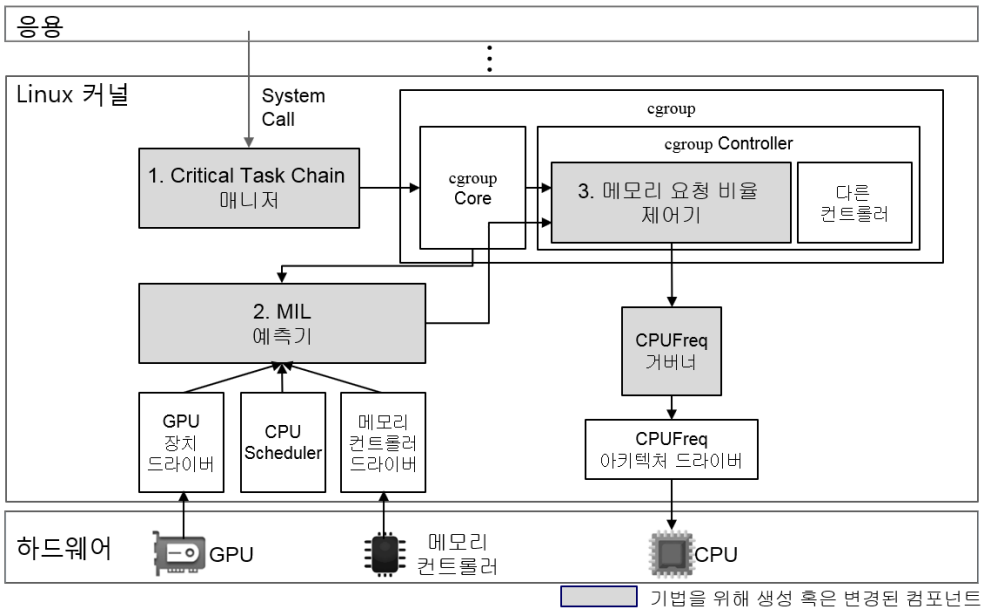

그림 6. 제안하는 동적 메모리 요청 Throttlng 기법의 전체 구조

1. Critical Task Chain 매니저
2. MIL 예측기
3. 메모리 요청 비율 제어기

Critical task chain 매니저는 critical task chain을 식별해 critical cgroup과 normal cgroup으로 나눠 관리하는 역할을 하고, MIL 예측기는 MIL을 주기적으로 예측해 메모리 요청 비율 제어기에 전달한다. 메모리 요청 비율 제어기는 각 core에서 수행되는 task의 criticality와 MIL 예측 내용을 바탕으로 CPUFreq 거버너에게 frequency제어 명령을 내려 normal cgroup의 메모리 요청 비율을 throttling한다. 메모리 요청 제어기를 cgroup controller에 추가하여 cgroup core에 의해 그룹화된 그룹에 연동되어 CPU frequency를

제어할 수 있게 하였다. 다음 4장에서 위 세가지 장치의 구체적인 역할에 대해 설명하도록 하겠다.

제 4 장 동적 메모리 요청 Throttling 기법

본 장에서는 제안하는 동적 메모리 요청 throttling 기법에 대해 상세하게 설명한다. 제안하는 기법은 세가지 핵심 컴포넌트들로 구성된다. 1절에서는 첫번째 컴포넌트인 Critical Task Chain 매니저에 대해서 설명하고 2절에서는 MIL 예측기에 대해 설명하고 마지막 3절에서는 메모리 요청 비율 제어기에 대해 설명한다.

제 1 절 Critical Task Chain 매니저

Critical Task Chain 매니저는 시스템내의 태스크를 critical task group과 normal task group으로 분류한다. 이를 위해 다음과 같은 두가지 핵심 동작을 수행한다.

1.1 Critical task chain 식별

Critical task chain 처음과 끝을 식별하기 위해 두가지의 라이브러리 함수를 새로 디자인하였다. 그림 7처럼 새로 생성한 함수 중 `start-of-critical-execution` 함수는 critical task chain에서 시작이 되는 태스크를 인식하고 critical task chain 매니저를 활성화한다. 사용자는 중요하다고 판단되는 동작에 해당하는 코드 섹션의 시작부분에 해당 함수를 명시적으로 넣음으로써 Critical task chain 식별을 시작하게 한다. `start-of-critical-execution`함수가 불리면 `write()` 시스템 콜 의해 critical task chain 매니저가 활성화된다.

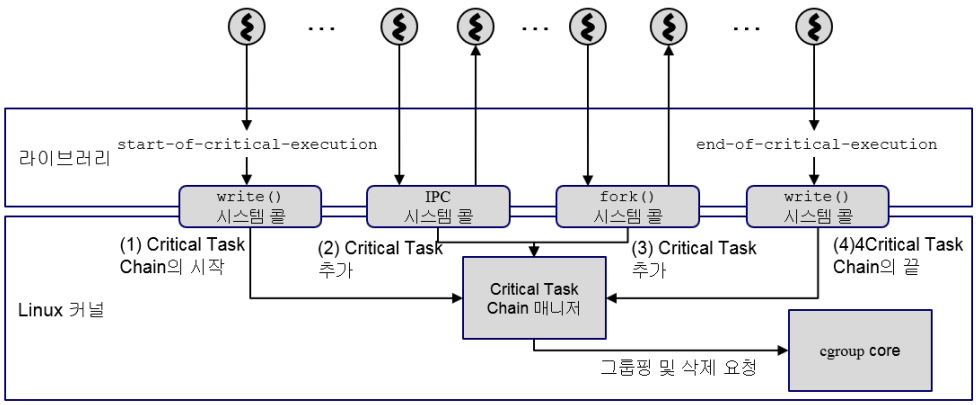

그림 7. Critical Task Chain 식별하는 시퀀스

critical task chain 매니저는 커널 자료구조에 critical task chain의 고유의 식별자를 저장하고, cgroup core에 그룹화 요청을 해서 start-of-critical-execution 함수를 호출한 태스크를 critical cgroup에 삽입한다. 새로 생성한 함수 중 end-of-critical-execution 함수는 critical task chain의 끝을 나타낸다. Critical task chain 매니저는 커널 자료구조에 critical task chain의 고유의 식별자를 지우고 해당 critical task chain의 태스크들을 critical cgroup에서 삭제 요청한다.

새로 디자인한 두 함수가 불리는 중간에 새로 생성되거나 IPC에 의해 서로 통신하는 태스크들도 중요 응용의 수행시간에 영향을 미치므로 critical task chain에 넣고 관리해야 한다. 이를 위해 두가지 시스템 콜 함수를 수정하였다. 하나는 IPC를 수행하는 시스템 콜이고 나머지 하나는 새로운 태스크를 생성하는 fork() 시스템 콜이다. 새로 디자인한 함수와 마찬가지로 critical task chain 매니저를 활성화해서 해당 태스크를 critical cgroup에 추가하도록 한다.

1.2 cgroup core에 group요청

cgroup core는 critical task chain에 속하는 task를 critical task group으로 관리한다. critical task chain 식별 시 필요한 네 가지 이벤트를 통해 critical task chain 매니저가 활성화된다. 활성화된 critical task chain 매니저는 네 가지 이벤트에 의해 critical task group으로 지정된 태스크 식별자를 cgroup의 /opt/mem-throttling/critical_cgroup 파일 시스템 디렉터리에 삽입하여 cgroup core에게 관리를 요청한다.

제 2 절 MIL 예측기

MIL 예측은 수행중인 태스크 정보와 outstanding memory request수를 이용해 표 1의 정책을 바탕으로 MIL 발생을 예측한다. Outstanding memory request란 메모리 컨트롤러 내의 메모리 요청 버퍼에 아직 처리되지 않은 메모리 요청이다. MIL 예측 정책은 총 세가지 경우로 나뉜다. 첫째 시스템 내에 태스크 $\tau \in C$ 와 태스크 $\tau \in N$ 가 동시 수행 중에 있고 outstanding memory request수가 $O_{Threshold}$

표 1. MIL 예측기 정책 설명

	태스크 정보	Outstanding memory request 수
MIL 발생	태스크 $\tau \in C$ 와 태스크 $\tau \in N$ 가 동시 수행 중	$O_{Threshold}$ 이상
MIL 미 발생	태스크 $\tau \in C$ 와 태스크 $\tau \in N$ 가 동시 수행 중	$O_{Threshold}$ 이하
	태스크 $\tau \in N$ 만 수행 중	상관 없음

이상이면 MIL이 발생되었다고 예측한다.

둘째 시스템 내에 태스크 $\tau \in C$ 와 태스크 $\tau \in N$ 가 동시 수행 중에 있더라도 outstanding memory request수 $O_{Threshold}$ 이하이면 MIL이 미 발생되었다고 예측한다. 셋째 시스템 내에 수행중인 태스크가 $\tau \in N$ 만 있다면 $O_{Threshold}$ 에 상관없이 MIL이 미 발생되었다고 예측한다. 지금부터 수행중인 태스크 정보를 확인하는 방법과 outstanding memory request를 통해 메모리 경합을 예측하는 이유를 설명한다.

2.1 수행중인 태스크들의 정보(criticality) 확인

MIL을 예측하기 위해서는 현재 시스템 내에서 수행중인 태스크의 criticality를 알아야 한다. critical task group에 속한 태스크가 수행중이지 않으면 메모리 요청 비율을 조절할 필요가 없기 때문이다. 정보를 확인하는 방법은 CPU에서 수행중인 태스크를 확인하는 방법과 GPU에서 수행중인 태스크를 확인하는 방법으로 크게 두가지로 나뉜다.

CPU에서 수행중인 태스크를 확인하려면 컨텍스트 스위칭(context switching) 시 변경되는 태스크가 critical task group에 속해 있는지 확인하면 된다. Linux의 `context_switch()` 함수는 OS 스케줄러에 의해 호출된다. 스케줄러는 정책에 의해 현재 수행중인 태스크를 대기 큐에 넣고 다음에 수행될 태스크를 대기 큐에서 삭제 후 수행시키는 작업을 하는데, `context_switch()` 함수는 태스크가 교체되기 전에 CPU 레지스터에 존재하던 이전 태스크의 컨텍스트를 다음 태스크의 컨텍스트로 교체하는 작업을 담당한다. CPU에서 태스크가 교체되는 모든 시점에 `context_switch()` 함수가 호출되므로 현재 수행 중인 태스크의 정보를 확인하는 용도로 사용하기에 적합하다.

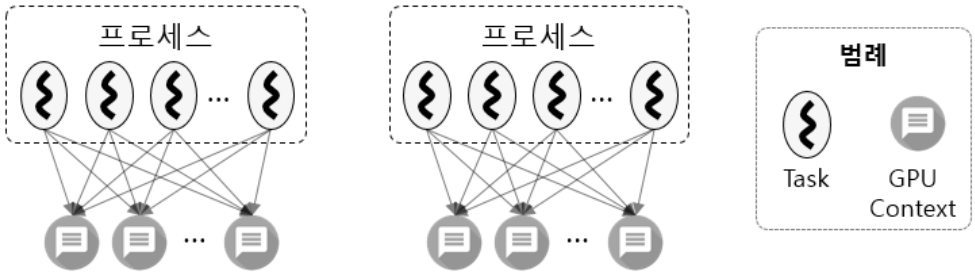

그림 8. 프로세스, 태스크와 GPU 컨텍스트 대응 관계

`context_switch()` 함수를 수정해서 현재 수행중인 태스크의 정보를 확인할 수 있도록 하였다.

GPU에서 수행중인 태스크를 확인하는 작업은 현재 가장 많이 사용되는 NVIDIA GPU의 정보를 공개하지 않는 자세로 인해 쉽지 않다. GPU에서 현재 수행중인 태스크를 정확히 확인할 방법은 현재 없다. 그래서 간접적으로 확인하는 방법을 취했다. GPU에서 태스크를 수행시키려면 GPGPU 라이브러리에서 제공하는 GPU 컨텍스트를 GPU 장치 드라이버를 통해 GPU로 보내야한다. GPU 컨텍스트는 GPU에서 수행되는 함수(보통 커널이라고 하는데 OS 커널과 혼동을 막기 위해 GPU에서 수행되는 함수로 지칭함)의 수행에 필요한 환경이다. GPU 컨텍스트에서 현재 GPU에서 수행중인 작업을 요청한 task의 식별자를 확인할 수 있다. 하지만 GPU 컨텍스트는 하나의 프로세스내의 여러 개 쓰레드에 의해서 중복 지정될 수 있어서 정확히 어떤 태스크가 수행중인지는 알 수 없다. 그래서 GPU 컨텍스트를 공유하는 태스크들 중 critical task group에 속한 태스크가 있는지 확인한다. 하나의 GPU 컨텍스트는 하나의 프로세스 내에서만 지정되므로 GPU 컨텍스트로부터 확인된 태스크들 중에 critical task group에 속한 태스크가 있으면, 확인된 태스크들 모두는 중요 응용을 수행하기 위한 태스크임은 분명하다.

2.2 Outstanding Memory Request를 통한 메모리 경합 예측

Outstanding memory request는 메모리 컨트롤러에서 메모리 요청 버퍼에 대기중인 메모리 요청이다. critical task group에 속한 태스크가 보낸 메모리 요청이 앞서 대기중인 메모리 요청이 많아 처리가 지연되면 critical task chain의 수행도 지연된다. 본 절에서는 outstanding memory request 수와 end-to-end latency와의 관계를 확인하고 현재의 outstanding memory request수를 통해 미래의 메모리 경합을 예측할 수 있는지 다음 세가지 관찰을 통해 확인한다.

- Outstanding memory request수와 end-to-end latency는 비례관계
- 메모리 접근 패턴의 지역성
- 메모리사용 집중 영역내 Outstanding memory request수 高-低가 교대로 나타남

세가지 관찰 중에 첫째 outstanding memory request수와 응용의 end-to-end latency의 관계를 알아보기 위해 실험을 진행하였다. 워크로드는 SPEC2006 벤치마크 프로그램과 메모리 요청을 다양하게 발생시키는 인조의 워크로드이다. 독립 변수는 outstanding memory request수이다. 인조의 워크로드의 메모리 요청 비율을 변화시켜가며 outstanding memory request수와 SPEC2006 벤치마크 프로그램의 end-to-end latency를 측정하였다. 결과는 그림 9와 같이 outstanding memory request수와 end-to-end latency가 비례관계임을 확인할 수 있었다. SPEC2006 벤치마크의 세 개 프로그램 중 메모리 요청 강도에

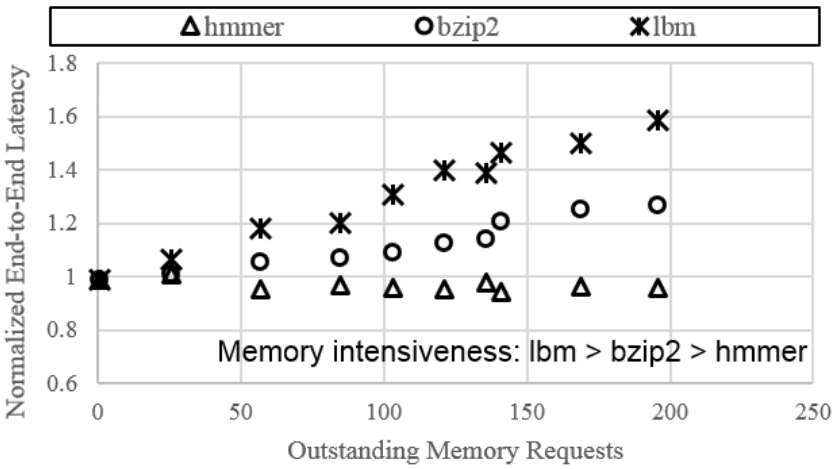

그림 9. SPEC2006 벤치마크 프로그램의 End-To-End Latency

따라 end-to-end latency는 증가함을 볼 수 있었다. 이는 메모리를 많이 사용하는 응용일수록 메모리 경합에 더욱 취약하다는 증거이다.

두번째 관찰은 그림 10의 실험을 통해서 확인하였다. 그림 10을 살펴보면 한 번 outstanding memory request수가 증가하면 수에서 수십 마이크로의 시간동안 유지되는 것을 확인 할 수 있다. 이는 메모리 접근 패턴이 지역성을 갖기 때문이다. 메모리에 한번 접근하면 캐쉬 라인크기만큼 한번에 캐쉬에 로드 한다. 캐쉬에 로드 된 영역에 접근할 경우 메모리 요청이 발생하지 않는다. 반면 캐쉬 메모리에 로드 되지 않은 영역을 처음 접근할 때는 많은 메모리 접근이 발생하게 된다. 그래서 그림 10과 같이 군집성으로 outstanding memory request가 증가하게 된다. 이는 스트림 데이터를 처리할 때 두드러지게 나타난다. 스트림 데이터 처리 시 보통 특정 양의 데이터를 저장해 놓고 한 번에

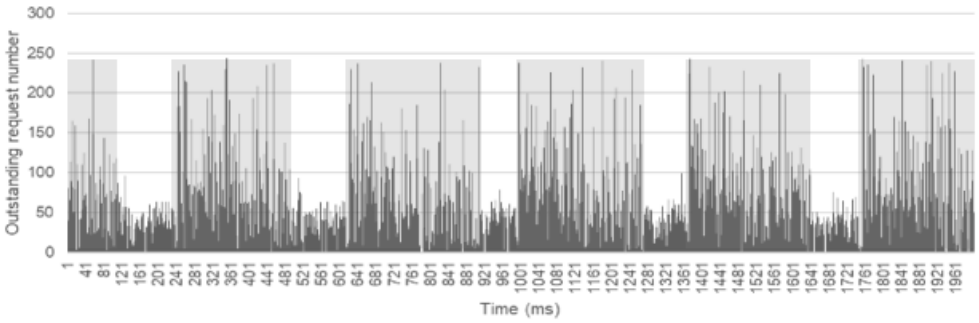

그림 10. YOLO 오브젝트 디텍션 응용 수행 시간별 outstanding memory request수 측정

처리하는 동작을 반복한다. 입력으로 들어오는 특정 양의 데이터를 저장할 때 메모리 접근이 많이 발생한다. 하지만 연산을 위해 접근할 때는 캐쉬에 한 번 로드 된 캐쉬 라인 사이즈의 데이터는 메모리 접근 시간이 줄어든다. 본 논문에서는 위 두가지 관찰을 통해 outstanding memory request가 메모리 경합을 예측하는 수단으로 사용될 수 있음을 확인하였다.

세번째 관찰은 그림 11와 같은데 예측력을 더욱 높이고 오버헤드를 줄이는 정책의 개발로 이어졌다. 메모리 접근 지역성 및 캐쉬 메모리 효과로 인해 메모리 접근이 집중된 영역에서의 outstanding memory request수가 항상 높지 않다는 것이다. 그림 9의 (2)번 차트는 주기적으로 outstanding memory request수를 측정했을 때 n번째 측정치와 n+1번째 측정치의 상관관계를 나타내는데 전체 중 항상 높은 경우는 10.5%밖에 되질 않고, 높고 낮음이 번갈아 있는 경우가 26.8%정도 차지했다. 그림 10의 (1)번 차트에서 보다시피 메모리 접근이 집중된 영역에서 높고 낮음이 반복된다. 메모리 집중영역내에서는 $O_{Threshold}$를 계속 초과해야 CPU frequency가 낮은 상태로 유지되는데 그림과 같으면 거버너의 frequency 변경 동작이 계속 반복되어 오버헤드가 커진다. 그리고 높고 낮음이 반복되는 이유는

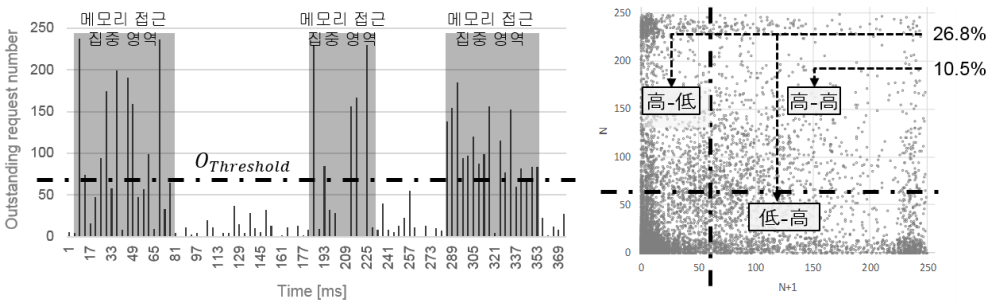

그림 11. (1) 메모리 접근 집중 영역에서의 outstanding memory request수의 패턴, (2) (1)의 N번째와 N+1번째 측정치 상관관계

실제로 메모리 요청의 수가 변경되는 경우도 있지만, 이미 요청으로 꽉 찬 메모리 요청 버퍼가 비워지는 것을 기다렸다가 비워진 후 버퍼로 들어가기 전에 측정되어서 낮은 것으로 추정된다. 일시적으로 낮아진 요청 수에 따라 frequency를 높이게 되면 예측력도 그만큼 떨어지게 된다. 그래서 MIL 예측을 한 번의 outstanding memory request 측정값으로 하지 않고 최근 두 번 측정된 값 중 한 번이라도 $O_{Threshold}$ 를 넘으면 메모리 경합이 발행했다고 예측한다. 한 번 측정치로 예측한 결과와 두 번 측정 치로 예측한 결과의 성능 개선 량 차이는 5장에서 실험을 통해 보였다.

ALGORITHM: MILPREDICTION algorithm

Data: The numper of currentOutstandingMemoryRequest : n
Data: The numper of beforeOutstandingMemoryRequest : m
Data: The numper of CPUcores : l

MILPREDICTION(*l, m, n*)

```
criticalCPU[ ] ← 0
criticalGPU ← 0
for i ← 1 upto l
    τ_i ← CURRENTTASKCPU()
    if (τ_i is critical) then
        criticalCPU[i] ← 1
    else
        criticalCPU[i] ← 0
end for
τ ← CURRENTTASKGPU()
if (τ is critical) then
    criticalGPU ← 1
else
    criticalGPU ← 0
if (all tasks is critical or all tasks is non_critical) then
    return no MIL
else if (n > O_threshold or m > O_threshold) then
    return MIL occurrence
else
    return no MIL
```

그림 12. MIL PREDICTION 알고리즘 의사 코드

제 3 절 메모리 요청 비율 제어기

메모리 요청 비율 제어기는 MIL 예측된 결과와 현재 CPU에 수행되는 태스크 정보를 바탕으로 CPU frequency를 throttling하는 역할을 한다. 태스크 정보는 cgroup core를 통해서 확인 가능하고 normal cgroup에 속한 태스크가 수행되는 CPU의 frequency를 조절한다. 메모리 요청 비율 제어기는 다음 두가지 동작을 수행한다.

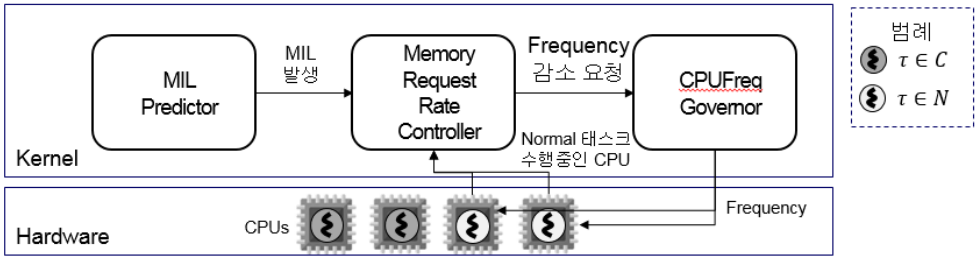

그림 13. 메모리 요청 비율 감소 요청 개념도

- 메모리 요청 비율 감소
- 메모리 요청 비율 원복

메모리 요청 비율 감소 동작은 그림 13과 같이 주기적으로 MIL 예측 결과를 바탕으로 진행된다. 주기적으로 MIL이 발생되었다고 예측되면 CPU 코어 중 normal task group에 속한 태스크가 수행중인 CPU 코어의 frequency를 낮춘다. 메모리 요청 비율 원복 동작은 MIL이 미 발생되었다고 예측되면 감소되었던 CPU의 Frequency를 높인다. 원복 동작은 주기및 비주기적으로 수행된다. 주기적인 동작은 정해진 주기마다 수행된다. 비 주기적인 동작은 CPU 컨텍스트 스위칭(context switching)시에 수행된다. 주기와 주기 사이에 CPU 컨텍스트 스위칭이 발생되면 수행 중인 태스크의 criticality가 변경될 수 있다. normal 태스크가 critical 태스크로 변경될 경우 감소되어 있던 frequency를 원복 하지 않으면 critical task chain의 end-to-end latency 증가를 초래한다. Frequency가 감소된 상황에서 컨텍스트 스위칭으로 인해 critical 태스크가 수행될 경우 메모리 요청 비율 원복 동작을 수행한다.

```
ALGORITHM: REQUESTTHROTTLING algorithm for CPUFreq governor

REQUESTTHROTTLING()
 1:   CPU_freq = CURRENTCPUFreq()
 2:   if (MIL occurrence = true) then
 3:       if(CPU_freq = freq_max) then
 4:           CPU_freq = freq_min
 5:       else
 6:           CPU_freq = CPU_freq
 7:   else
 8:       if(CPU_freq = freq_min) then
 9:           CPU_freq = freq_max
10:       else
11:           CPU_freq = CPU_freq
12:   return CPU_freq
```

그림 15. CPU Frequency Throttling 알고리즘 의사 코드

그림 15는 메모리 요청 비율 조절기에서 CPUFreq 거버너에게 동작 요청할 때 전달하는 CPU frequency값을 구하는 알고리즘이다. 먼저 Normal 태스크가 수행중인 CPU 코어의 현재 frequency를 확인한다. MIL이 발생했으면 frequency를 설정할 수 있는 가장 낮은 값으로 설정한다. 만약 발생하지 않았으면 frequency를 설정할 수 있는 가장 높은 값으로 설정한다.

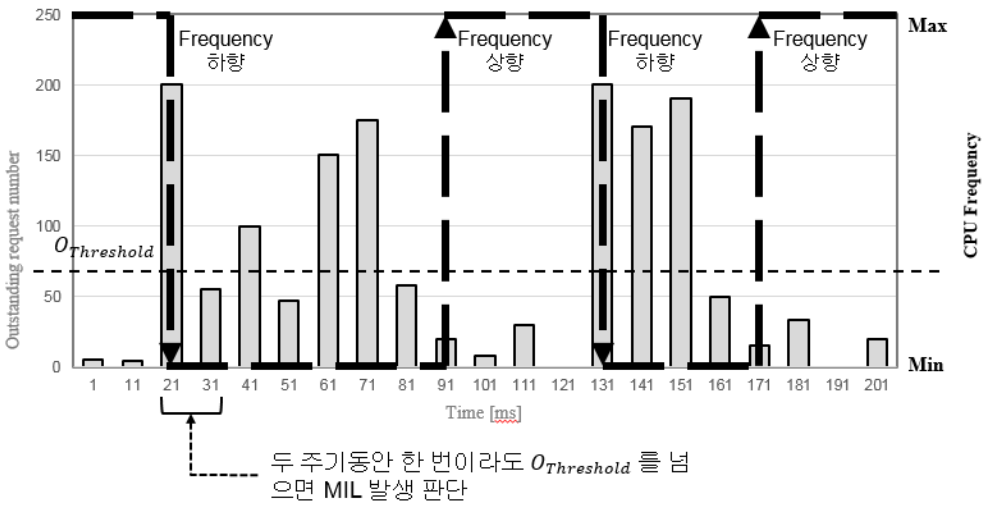

그림 14. 메모리 요청 비율 제어기 동작 예시

메모리 요청 비율 제어기의 동작은 그림 14와 같이 최근 두 주기 동안 한 번이라도 $O_{Threshold}$를 넘으면 CPU frequency throttling을 진행하고 그렇지 않으면 원복 한다.

제 5 장 구현과 실험적 평가

본 장에서는 제안한 동적 메모리 요청 throttling 기법을 실제 시스템에 구현하고 구 효과를 실험적으로 평가한 결과에 대해 설명한다. 다음과 같이 세 개의 절로 구성된다. 1절에서는 동적 메모리 요청 throttling 기법을 구현했을 때 발생한 특이사항에 대해 서술한다. 2절에서는 실험 설정에 대해 자세히 설명하고, 3절에서는 제안된 기법을 적용했을 때와 적용하지 않았을 때의 차이를 통해 그 효과를 실험적으로 입증한다. 그리고 기법 적용에 따른 오버헤드를 확인한다.

제 1 절 동적 메모리 요청 Throttling 기법의 구현

제안된 기법은 우분투 16.04, Linux 커널 4.4.38, CUDA 8.0으로 동작하는 NVIDIA Jetson TX2에 구현되었다. NVIDIA Jetson TX2는 2개의 Denver 코어로 구성된 CPU 클러스터와 4개의 A57 코어로 구성된 CPU 클러스터, 256개의 NVIDIA Pascal 코어로 구성된 GPU가 메모리 하드웨어를 공유하는 이기종의 CPU-GPU 아키텍처로 되어있다. 제안된 기법에서는 Linux의 `cgroup` core가 critical `cgroup`을 묶을 수 있도록 확장하고, `cgroup` 컨트롤러에 메모리 요청 throttling을 수행하는 메모리 요청 비율 제어기를 추가하였다. 그리고 실제 CPU의 동작 frequency를 조정하여 메모리 요청 비율 throttling을 수행하도록 CPUFreq 가버너를 수정하였다. 제안하는 기법은 크게 세가지 핵심 컴포넌트로 구성된다. 다음 절부터는 세가지 핵심 컴포넌트 구현 시의 특이사항에 대하여 설명한다.

1.1 Critical Task Chain 매니저 구현

Critical task chain에 들어갈 task를 식별하는 네 가지 이벤트 중에 IPC 시스템 콜에 의한 식별 방법을 구현하기 위해서는 IPC 핸들러 수정이 필요하다. Linux가 제공하는 IPC 메커니즘들 중 socket IPC 메커니즘을 우선적으로 선정하여 핸들러를 수정하였다. 가장 많이 사용되는 IPC handler 중 하나이고, 다양한 IPC 메커니즘을 지원한다. 향후에 RPC, Binder IPC와 같은 high-level IPC 메커니즘으로 확장될 수 있다.

`fork()` 시스템 콜에 의한 식별 방법을 구현하기 위해서는 IPC 핸들러 수정이 필요하다. `fork()` 핸들러는 부모 태스크가 critical 태스크이면 자식 태스크도 critical 태스크로 식별하도록 수정하였다. 수정된 `cgroup` core의 핸들러는 시스템에 critical 태스크가 새로 추가되거나 제거될 때마다 critical `cgroup`들과 normal `cgroup`의 태스크 식별자 리스트를 갱신한다.

1.2 MIL 예측기 구현

MIL 예측기 구현 시 수행중인 태스크 정보와 outstanding memory request수를 얻기 위해 Nvidia GPU 장치 드라이버에서 제공하는 API를 사용하였다. `gr_fecs_current_ctx_r()` API를 통해 GPU에서 수행 중인 GPU 컨텍스트를 식별하였다. 식별된 컨텍스트에서 태스크들을 식별하기 위해 `gk20a_fecs_trace_find_pid()` API를 사용하였다. 또한 Outstanding memory request수를 측정하기 위해 태그라(Tegra) SoC의 메모리 컨트롤러 드라이버가 제공하는 `mc_readl()` API를 사용하였다. `mc_readl()` API를 통해

MC_EMEM_ARB_OUTSTANDING_REQ 레지스터를 읽으면 현재 메모리 컨트롤러의 버퍼에 있는 Outstanding memory request수를 측정할 수 있다. 이렇게 측정이 불가능한 시스템에서는 PMU를 통해 LLC 미스(miss)율을 구함으로써 간접적으로 메모리 사용량을 측정할 수 있다.

1.3 메모리 요청 비율 제어기 구현

cgroup 컨트롤러에 메모리 요청 throttling을 수행하는 메모리 요청 비율 제어기를 추가하였다. 컨텍스트 스위칭 할 때 태스크의 criticality가 변경될 경우 메모리 요청 비율을 원복 할 수 있도록 context_switching() 함수를 수정하였다. Jetson TX2 에서는 두개의 CPU 클러스터가 존재한다. 한 CPU 클러스터 내의 코어들은 frequency가 연동된다. 따라서 CPU 클러스터 단위로 frequency를 조절하도록 하였다. 한 CPU 클러스터 내에 한 코어라도 critical 태스크를 수행한다면 이 CPU 클러스터가 critical 태스크를 수행 중이라고 판단한다. MIL 예측기와 메모리 요청 비율 제어기의 동작 주기는 10ms로 설정하였다. 이는 기존 CPUFreq 거버너와 같은 주기이다. 실험적으로 결정되었는데 10ms 보다 짧을 경우 CPUFreq 거버너 동작 오버헤드로 인해 성능이 감소된다.

제 2 절 실험 설정

본 절에서는 실험들의 설정에 대해 설명한다. 실험 환경을 표 2에 요약해 두었다. 본 실험은 앞서 설명한 대로 우분투 16.04, Linux 커널

4.4.38, CUDA 8.0으로 동작하는 NVIDIA Jetson TX2에 수행되었다. 총 세가지 실험을 진행하였다. 첫번째 실험은 outstanding memory request의 적절한 threshold를 알아보는 실험이다. 두번째 실험은 제안된 기법의 효과 측정 실험이다. 마지막 세번째 실험은 런타임 오버헤드를 측정하는 실험이다.

첫번째 실험의 목적은 critical 응용의 성능 개선 대비 normal 응용의 성능 열화 trade-off 관계 확인을 통한 outstanding memory request threshold 가이드를 위한 실험이다. 실험에서는 두 응용을 동시에 수행시켰을 때 두 응용의 수행시간을 outstanding memory request수 별로 확인한다. Normal 응용은 인조의 워크로드로써 메모리 요청을 발생시킬 수 있는 한도내에서 최대로 발생시키도록 설계하였다. 이는 가혹한 환경에서 normal 응용의 수행시간이 얼마나 감소되는지 확인하기 위함이다. Critical 응용은 YOLO라는 딥러닝 기반 오브젝트 디텍션을 수행하는 응용을 선정하였다. 이 응용의 end-to-end latency는 이미지 한 장을 입력으로 받은 순간부터 오브젝트 디텍션(object detection) 결과를 디스플레이에 출력하는데 소요되는 시간이다. YOLO는 ILSVRC(ImageNet Large Scale Visual Recognition Challenge) 대회에서 상위 랭킹에 위치한 많은 오브젝트 디텍션

표 2. 실험 설정 요약

Hardware	CPU	Denver2 dual cores, A57 quad cores
	GPU	NVIDIA Pascal 256 cores
	Memory	8GB 128bit LPDDR4 (59.7GB/s)
	Storage	32GB eMMC
Software	GPGPU Library	CUDA 8.0
	Framework	Ubuntu 16.04
	Kernel	Linux Kernel Version 4.4.38

s알고리즘들에 기초로 사용될 정도로 빠른 오브젝트 식별 시간과 정확도를 가지는 최신의 오브젝트 디텍션 응용이다. 본 실험에서 사용된 YOLO 버전은 가장 최신의 tiny, 2, 3 이다. YOLO 응용을 수행하기 전에 critical task chain식별을 위해 시작과 끝에 `start-of-critical-execution, end-of-critical-execution` 함수를 명시하였다.

두번째 실험의 목적은 인조의 워크로드로 발생시키는 메모리 요청 비율별로 제안된 기법에서 critical 응용의 성능 개선 량과 normal응용의 성능 저하 정도를 파악하기 위함이다. 우선 Critical 응용의 성능 개선 량 확인을 위해 직접 설계한 인조의 normal 응용과 critical 응용을 동시에 수행시켜 critical응용의 end-to-end latency를 측정하였다. 인조의 워크로드는 메모리 요청을 0~100%까지 발생시킬 수 있도록 설계되었다. Critical 응용으로는 첫번째 실험과 같은 YOLO 응용을 사용하였다. 실험은 메모리 요청 비율 별로 모든 구간에서 개선 율 및 특이 사항을 확인하기 위해 인조의 워크로드의 메모리 요청 비율을 변경시키며 critical 응용의 end-to-end latency를 측정하였다. End-to-end latency는 이미지 한 장을 입력으로 받은 순간부터 오브젝트 디텍션 결과를 디스플레이에 출력하는데 소요되는 시간이다. 실험의 정확도를 위해 1000장의 서로 다른 프레임의 end-to-end latency를 측정하여 그 평균값으로 결과를 도출하였다. 다른 하나의 실험은 normal 응용의 성능 저하를 확인하는 실험이다. 성능 저하는 기법 적용 전후의 스트리밍 동영상의 FPS 드랍 율로 확인하였다. 스트리밍 동영상은 대표적인 인포테인먼트 응용 중 하나이다. 보통 30 FPS 정도 성능에서 사용자들은 인지 상의 끊김없이 동영상 시청을 할 수 있다. 때문에 구글 유튜브 스트리밍 동영상 재생 시 30 FPS의 성능이 나온다. 실험은 YOLO응용과 구글 유튜브 스트리밍 동영상을

동시에 수행 시 동영상의 FPS를 측정해서 30 FPS 대비 저하되는 양을 기록하였다. FPS 측정은 구글 크롬에서 제공하는 성능 측정 도구를 사용하였고, 한 프레임의 해상도는 854x480으로 설정되었다.

세번째 실험의 목적은 제안된 기법 수행 시 발생되는 런타임 오버헤드를 측정하는 것이다. 제안된 기법의 세가지 핵심 컴포넌트가 주기적으로 반복해서 실행될 때 발생되는 오버헤드를 측정하였다. 실험은 가장 가혹한 조건인 메모리 요청 비율 100%를 발생시키는 인조의 워크로드에서 YOLO 응용을 동시에 수행하며 측정하였다.

제 3 절 실험 결과

본 절에서는 앞에서 설명한 세 가지 실험을 진행한 결과를 설명한다.

3.1 실험1: Outstanding memory request의 적정 threshold 확인

본 실험에서는 $O_{Threshold}$ 별로 normal 응용과 critical 응용을 동시에 수행시키며 end-to-end latency의 증가 량을 확인하였다. 최대로 설정할 수 있는 $O_{Threshold}$ 는 255이고 실험은 그 값의 10%~85%까지 15% 단위로 증가시키며 진행하였다. 그림 16에서 실험 결과를 차트로 보여준다. $O_{Threshold}$ 가 낮을수록 normal 응용의 성능 열화가 기하 급수적으로 증가하였다. 가장 낮은 10%의 $O_{Threshold}$ 에서는 YOLOv3의 end-to-end latency가 2.4배 증가하였다. 또한 너무 높을 경우 critical 응용의 성능 개선 량이 감소해 기법 적용의 필요성이

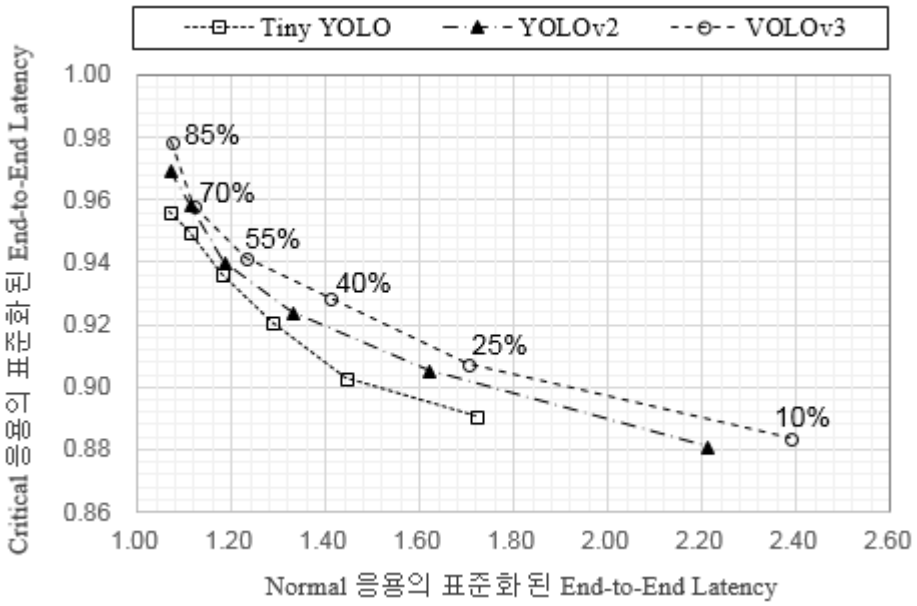

그림 16. $O_{Threshold}$별 Normal 응용과 Critical 응용의 성능 비교

없어진다. $O_{Threshold}$는 이와 같이 trade-off 관계가 있으므로 normal 응용의 성능이 감내 되는 범위에서 적절하게 설정해야 한다. 너무 낮으면 critical 응용의 성능 개선 량 대비 normal응용의 성능 감소량이 너무 크기 때문에 25%이상 설정할 것을 권고한다. 다음 실험은 $O_{Threshold}$를 25%로 설정하고 진행하였다.

3.2 실험2: 제안된 기법의 성능 개선 량 확인

표 3. 제안된 기법의 성능 개선 량

	메모리 요청 비율	Critical 응용														
		Tiny YOLO					YOLOv2					YOLOv3				
		기존		변경		개선율	기존		변경		개선율	기존		변경		개선율
		평균	표준편차	평균	표준편차		평균	표준편차	평균	표준편차		평균	표준편차	평균	표준편차	
	(%)	(ms)		(ms)		(%)	(ms)		(ms)		(%)	(ms)		(ms)		(%)
Critical 응용의 E2E Latency	20	54.73	±2.11	54.11	±2.69	1.14	139.98	±0.94	137.69	±1.78	1.66	297.50	±0.59	292.58	±1.06	1.68
	40	57.41	±2.14	55.00	±2.81	4.38	145.90	±1.14	140.13	±2.14	4.12	308.10	±1.11	296.21	±1.48	4.01
	60	60.59	±2.12	56.31	±3.23	7.59	153.34	±1.29	143.01	±2.88	7.22	315.44	±1.12	301.97	±2.65	4.46
	80	62.30	±2.54	57.44	±3.26	8.47	158.53	±1.37	144.56	±3.04	9.66	331.87	±1.24	305.66	±2.88	8.58
	100	62.85	±2.46	57.19	±3.32	9.89	159.33	±1.14	144.05	±3.07	10.60	333.81	±1.47	305.24	±2.97	9.36

첫번째 실험에서는 인조의 워크로드가 발생시키는 메모리 요청 비율별로 critical 응용의 성능 개선 량을 확인하였다. 성능 개선 량은 YOLO응용 수행 시 처리되는 1000장의 프레임의 end-to-end latency 평균을 통해 확인하였다. End-to-end latency는 한 장의 프레임이 처리되어 디스플레이 되는데 소요되는 시간이다. 결과는 표 3에 보이는 것처럼 normal 응용의 메모리 요청 비율이 증가함에 따라 개선 량도 증가하는 것으로 확인 되었다. 개선 량은 최대 10.6%였고 그때 개선 기법 적용 측정 결과의 표준 편차는 ±2.13%정도였다.

두번째 실험에서는 normal응용의 성능 저하 정도를 스트리밍 동영상의 FPS 드랍 율을 통해 확인하였다. YOLO(tiny)응용과 동영상을 동시 수행 시 30 FPS 대비 저하되는 FPS 비율을 FPS 드랍 율로 정의하고 측정하였다. YOLO응용과 동시에 동영상 한 개를 재생 시 FPS 드랍 율은 기법 적용 전이 2.5%, 적용 후가 5%로 동영상 시청에 큰 무리가 없는 성능이었다. 하지만 동영상을 두 개 재생 시에는 기법 적용 전의 FPS 드랍 율이 25%, 적용 후가 42%로 기법 적용 전 후 모두 인지상의 동영상 끊김이 발생하는 수준이었다. YOLO응용과

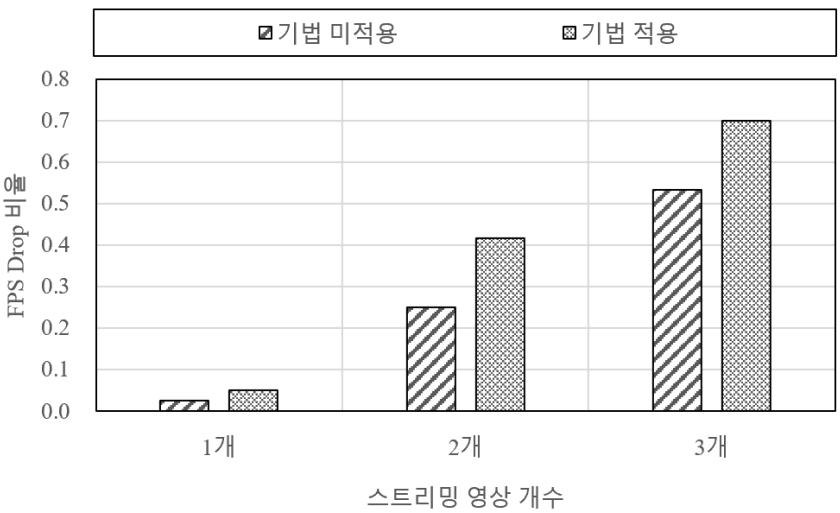

그림 17. 스트리밍 동영상의 FPS 드랍 율

인포테인먼트을 동시 수행 시 적당한 인포테인먼트 응용의 수행은 기법을 적용하더라도 큰 문제가 없지만, 과도한 인포테인먼트 응용의 수행은 기법 적용여부과 관계없이 수행에 무리가 있음을 확인 하였다.

3.3 실험3: 제안된 기법의 런타임 오버헤드 확인

본 실험에서는 제안된 기법의 세가지 핵심 컴포넌트를 수행 시 발생되는 오버헤드를 측정하였다. 주기적으로 MIL 예측기 및 메모리 요청 비율 제어기가 수행되므로 구현상의 오버헤드가 크면 성능 효과가 떨어진다. Normal application에서 발생시킬 수 있는 가장 큰 비율의 메모리 요청 워크로드 하에서 YOLO응용 세가지를 수행시켰을 때의 오버헤드는 최대 1.74%로 미비한 수준이다. 측정은 하나의 도로 영상이 모두 완료될 때까지 약 3분 가량 진행되었고 제안된 기법이 10ms주기로 약 5만 4천번 가량 수행되었다. 측정된 값은 그 평균 값이다.

표 4. 제안된 기법의 런타임 오버헤드

	Tiny yolo	Yolo v2	Yolo v3
런타임 오버헤드 (%)	1.74%	1.43%	1.5%

제 6 장 결 론

본 논문에서는 메모리 경합에 의한 중요 응용의 end-to-end latency 지연 문제를 해결하는 애플리케이션 인식 기반 동적 메모리 요청 비율 throttling 기법을 제안하였다. 본 기법은 시스템내 수행 중인 태스크들을 critical 태스크와 normal 태스크로 분리하는 사용자 라이브러리 함수를 제공하고 함수 사이에 새로 생성되거나 통신하는 태스크를 모두 critical 태스크로 분리한다. 분리된 그룹은 cgroup을 관리되고 cgroup 컨트롤러를 통해 제어된다. 주기적으로 메모리 경합이 발생했는지 예측하고 경합이 발생했을 경우 CPUFreq 거버너의 CPU frequency 조절을 통해 normal cgroup의 메모리 요청을 제한 한다.

제안하는 기법의 효과에 대한 실험 결과 critical 응용의 end-to-end latency가 기존 대비 10.6%까지 감소되는 것을 확인하였다. 그에 따른 오버헤드는 1.76%로 크지 않은 수준이었다. 그리고 outstanding memory request의 threshold실험에서는 normal 응용의 성능 저하가 $O_{Threshold}$ 가 낮을수록 기하 급수적으로 증가하였다. Normal 응용의 성능을 개선할 필요가 있다면 MIL 예측력을 더욱 높여 필요한 구간에만 throttling을 해서 불필요한 throttling을 피하거나, 메모리 요청의 강도가 높은 normal응용을 critical 응용과 동시에 수행되지 않도록 스케줄링 하는 연구가 진행되어야 할 것이다. 제안하는 기법으로 critical 응용의 메모리 경합으로 인한 성능저하를 감소시킬 수 있다는 것이 확인 되었으므로, 다양한 시스템에 사용되길 기대한다.

참 고 문 헌

[1] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, S. Wang “An Evaluation of the NVIDIA TX1 for Supporting Real-Time Computer-Vision Workloads”, IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017.

[2] Article, http://electronicsmaker.com/multicore-architecture-brainpower-of-smart-car, 2016.

[3] R. Ernst, M. Di Natale, “Mixed Criticality Systems—A History of Misconceptions?”, IEEE Design & Test, 2016.

[4] C. Roberto, C. Nicola, B. Marko, “Memory Interference Characterization between CPU cores and integrated GPUs in Mixed-Criticality Platforms”, IEEE International Conference on Emerging Technologies and Factory Automation (IEEE ETFA), 2017.

[5] H. Tejun, “Control group v2”, 2015

[6] Nvidia technical reference manual, “NVIDIA Parker Series SoC”, 2017.

[7] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, T. Moscibroda, “Reducing memory interference in multicore systems via application-aware memory channel partitioning”, Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2011.

[8] Y. Heechul, R. Mancuso, Z. Wu, R. Pellizzoni, “PALLOC: DRAM bank-aware memory allocator for performance isolation on

multicore platforms", IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014.

[9] L. Lei, Z. Cui, M. Xing, Y. Bao, M. Chen, C. Wu "A software memory partition approach for eliminating bank-level interference in multicore systems", 21st international conference on Parallel architectures and compilation techniques. ACM, 2012.

[10] I. Hur, C. Lin, "Memory scheduling for modern microprocessors", ACM Transactions on Computer Systems, 2007.

[11] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, J. Oh, "A predictable and command-level priority-based DRAM controller for mixed-criticality systems", IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015.

[12] Y. Kim, M. Papamicheal, O. Mutlu, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior", Proceedings of the 43th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2010.

[13] O. Mutlu, T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems", Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA), 2008.

[14] O. Mutlu, T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors", Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture. IEEE Computer Society, 2007.

[15] S. Rai, M. Chaudhuri, "Using Criticality of GPU Accesses in Memory Management for CPU-GPU Heterogeneous Multi-Core

Processors", ACM Transactions on Embedded Computing Systems (TECS), 2017.

[16] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, L. Sha "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms", Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013.

[17] H. Yun, W. Ali, S. Gondi, S. Biswas, "BWLOCK: A dynamic memory access control framework for soft real-time applications on multicore platforms", IEEE Transactions on Computers, 2017.

[18] CUDA Toolkit Documentation, https://docs.nvidia.com/cuda/, v9.2.88.

[19] T. Amert, N. Otterness, M. Yang, J. H. Anderson, F. D. Smith, "GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed", Real-Time Systems Symposium (RTSS), 2017.

[20] W. Ali, H. Yun, "Work-In-Progress: Protecting Real-Time GPU Applications on Integrated CPU-GPU SoC Platforms", IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017.

[21] S. Huh, J. Yoo, S. Hong, "Cross-layer Resource Control and Scheduling for Improving Interactivity in Android", International Journal of Software: Practice and Experience, 2015.

[22] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, C. Kozyrakis, "Improving Resource Efficiency at Scale with Heracles", ACM Transactions on Computer Systems (TOCS), 2016.

Abstract

Application-Aware Dynamic Memory Request Throttling to Reduce Memory Interference Latency

Daesik Ham
Department of Transdisciplinary Studies / Intelligent Systems
The Graduate School of Convergence Science and Technology
Seoul National University

Modern multi-core processors are rapidly being adopted in a variety of mission-critical embedded systems because the latest architecture can achieve high performance within a limited power budget. In this mission-critical embedded system, many dedicated low-end microcontrollers that perform some specific tasks are integrated into a few high-performance multicore processors. This integrated mission-critical embedded system requires management of resource contention for applications sharing various computing resources. If the management of resource contention is not effective, mission-critical applications may be delayed due to an interference with other applications, leading to a serious accident.

One of the most common examples of embedded systems with critical applications is InfoADAS. ADAS is an advanced driver assistance system in which critical applications are performed primarily, while infotainment is a normal application that only provides information and convenience to passengers. There are many

data-intensive applications in ADAS that process large amounts of data collected from cameras, radars, and LiDARs to obtain information. In a system that performs such an application, it is essential to effectively restrict memory contention.

In this paper, we propose an application-aware dynamic memory request rate throttling technique to solve the performance delay problem of critical applications by memory contention. This technique classifies the applications in the system into critical applications and normal applications, and then manages them as `cgroups`. It predicts whether there is a memory contention periodically and throttles the memory request rate of the normal application by changing the CPU frequency by CPUFreq governor. To verify the effectiveness of our technique, we implemented on the NVIDIA Jetson TX2 board with Linux kernel 4.4.38. Experimental results show that the performance improvement of 10.6% is higher than before, while incurring only negligible overhead.

Keywords: memory contention, memory interference, CPU-GPU heterogenous multicore, CPU throttling, mixed-criticality system

Student Number: 2016-26589